PoleX DevOps Team



Dive into OpenStack Deployment

with PuppetOpenStack

O RLY?

Xingchao Yu

目錄

关于本书

人是很奇怪的动物，天生懒惰，但是在压力的驱使下，又能做成一些连自己都会觉得惊讶的成就。对于天生喜欢写写随笔和技术blog的我，在近三年的时间里，虽有多家出版社编辑的循循善诱，仍然不为所动，心想着有这闲时间我不如多吃几份美食，多打两盘DOTA。

更深层次地讲其实是源于对于未知的恐惧，源于面对数十万文字工作量的害怕，这本就是一场体力和脑力并存的马拉松，普通人在看到终点的艰巨时，在起点就已经选择了放弃，然而我不是一个普通的人，因此早在6个月前我就富有先见之明地创建了这本书，并且写上了长达83个字节的详细介绍，然后就没有了下文。

无数次惨痛的教训告诉我：决心虽然重要，但是坚持更加可贵。让我重新拾起笔（键盘）的原因是源于今晚的一顿饭。今晚注定是一个平凡的夜晚，DevOps组的同学们像往常一样忙到晚上8点多已是饥肠辘辘，正纷纷收拾东西下班，大家站在公司门前讨论去哪里吃饭，这时我走出来，虽刚从纽约飞回到空气清新的帝都，时差还没有倒回来，脑袋浑浑噩噩，但又担心这帮邋遢的家伙站在公司大门口持续影响公司形象，于是忙把他们连哄带骗全都打包上车，拉到了附近的一家馆子里，几口飘香的云南饭菜入肚，大家就开始天南地北地扯东扯西，无外乎是谁谁谁今天又看了一本MySQL从删库到跑路，谁谁谁今天又当了一回Puppet背锅侠，谁谁谁的项目又埋了一个大坑。

不知不觉，我们就聊到了OpenStack自动化部署这个话题上。作为一名从2011年开始接触Openstack的老油条，在13年幸运地混进了PuppetOpenstack社区项目成了一名core，这漫漫五年时间里有许多知识经验和教训值得沉淀，因此我深深地感觉到是时候~~忽悠~~召唤这帮孩子来一起填这个史诗级别的远古巨坑了。

DevOps Team从13年伊始就全身心投入到持续交付和持续集成事业中，目前使用了**96**个puppet module, **6**台PuppetMaster, 集中管理着约**87**个Openstack集群,**7**种不同环境，支撑了近**3500**台Openstack集群服务器。独立开发了云平台部署工具(Pluto)， 软件包管理工具(Packforge/Specforge/Repoforge)， IaaS虚拟资源池(Chameleon）, Openstack升级套件(Screenwriter)，这里面涉及到了大量的自动化运维工具，例如：Ansible, Foreman, Puppet, ClusterShell, Mcollective；同时还有大量运维脚本，包含了多种语言，如：Shell, Ruby, Python, Puppet。

说实话，DevOps团队里每个人平时都有忙不完的事情，从早上还没到公司到晚上回到家中，随时随刻会被人on call，就像今天吃饭那样，维宇同学正站在门口等我上个wc的时间，就被其他同事喊回去处理问题去了。但并不是说因此就有理由说，我们很忙，忙得没有时间写一本书。我们的确需要停下脚步，回头看一看过去三年的努力，然后做一个系统性的梳理和总结了。

本书是关于对Openstack自动化部署工作核心部分的讲解：

> PuppetOpenstack核心模块和基础模块的详细介绍和最佳实践

我们规划每天晚上抽出3小时以上的不固定时间，计划在2个星期内（210+人时）完成第一版的初稿工作，然后开始进入稳定的迭代周期。而这本书仅是一个起步，这几天我规划了接下来的技术分享输出的Roadmap，比如Orchestration，CI, Rolling upgrade等热门话题。我们不想雷声大雨点小，因此当这些事情在真正落到地面的时候，你会听到那些令人振奋的声音。

最后，希望关注此书的读者们可以从中有所思，也有所得。

于月光明媚昏昏欲睡的五月凌晨

余兴超

- 关于本书

概览

关于本书

本书的主旨是为读者讲解如何借助当前流行的自动化运维工具目来完成OpenStack云平台的部署和配置工作，本书大致划分为三大部分：

- 介绍部分 包含前期的准备工作，相关约定，术语说明，项目概览，模块剖析等等基础知识给读者从全局上的认识
- 配置管理部分 本部分分为三章，分别介绍Openstack使用到的基础模块和核心服务模块以及公共库模块的介绍
- 最佳实践部分 主要介绍在实际生产环境中应注意的细节和管理规约

关于讲与不讲

这点很重要，笔者在撰写本书前也曾设想编写一本OpenStack运维大全，涉及到运维中的所有环节，但发现其工作量和难度远远超出了事先的预估。延期在IT领域内是一件常识，因此在做一件事情前尚若没有考虑清楚其界限和范围，那么就容易引起工程延期甚至无法完成。因此，对于本书，其主要目的是：

> 讲解如何使用自动化配置管理工具Puppet完成OpenStack

我们不会讲的是：

核心服务模块的主要class和define，重要params，使用陷阱，注意事项，使用技巧；像reference books那样事无巨细地讲解每个模块的每个class,define,custom resource,facter，每个params的说明。因为我们不是超人，你也不是机器人。

关于OpenStack

Openstack目前已经成为开源IaaS项目的翘楚。在去年Openstack推出BigTent战略后，在Openstack名下的项目已经多达百个。那么在面对如此复杂的架构和众多服务，我们该如何去面对？

Fuel/Packstack/TriplO/Ctask和PuppetOpenstack的关系

- Packstack封装了PuppetOpenstack，使得用户在终端下可以通过交互式问答或者非交互式YAML格式文件的方式去部署Openstack集群，使得用户无需了解Puppet和PuppetOpenstack的细节。
- Fuel更进一步，提供了友好的Web UI界面，使得用户对于技术细节如何实现上做到了非常好的隐藏，还提供了一些健康检查工具，确保部署符合预期。
- TripleO使用Openstack的现有项目来部署Openstack，tripleo-puppet-elements组件用于生成部署Openstack的磁盘镜像文件，直接使用到了PuppetOpenstack。
- Ctask类似于Packstack，封装了PuppetOpenstack，不同点在于整合了内部开发的网络检查工具，分布式存储检查脚本，确保每步的输出符合预期，并能快速定位到问题的根源。

为什么要学习OpenStack自动化部署?

我们的目标读者是实施工程师，运维工程师，DevOps工程师和研发工程师，是一个不甘于只会使用工具的人，喜欢探索新的事物，喜欢去刨根问底。
同时现有基于PuppetOpenstack封装的S工具并不能100%满足用户的需求，如果你没有手动能力的话，那你只能采用一些很low的方法，比如使用Fuel部署了一套集群，然后再手动修改配置文件，手动重启服务！一周后，一个月后，你还能记住你当时做的操作吗？之后来维护的同事，他们知道你对这套复杂的软件栈做了什么吗？
No，在运维自动化的世界里，一切都应该自动的，不依赖于具体的人，而是依赖于稳定强大的自动化运维系统。
如果你是一名正在或者即将要做Openstack集群部署和管理的工程师，那么这就是你应该看的书籍。

和电子版本有什么不同？

最初我们在Gitbook上开始了本书的编写，获得了广泛的关注和评论。但是电子版本更像是一个collection，来自于多个协作者的共同产物，在内容统一和用词上没有做到严谨精确，当时所使用的Mitaka版本已经滞后于最新稳定版本近一年。更重要的是，在这一年多的时间里，我们在新上线的OpenStack集群上做了许多新的尝试和总结。

因此，纸质版会包含更多更新令人感兴趣的内容，包括对于网络的管理，操作系统的管理，运维节点的设计等等。

约定和说明

文章风格说明

因为本书共有5名参与编写的同学，为了保证行文分各个，我(xingchao)会负责概览部分和最佳实践部分的编写（果然是打酱油的命），同时也会负责部分模块章节的编写，其他同学则会专注在核心模块部分，给出各个复杂模块的详细介绍。

同时，我们每天会在微信群里~~唠嗑~~讨论，如何改进各个模块章节的书写方式和风格，并保持格式上的严格统一，给予读者更好的阅读体验，同时如果你在阅读过程中发现有错别字，不合理的说明，非常欢迎指正，我们会在第一时间更正。

文章格式说明

本文使用标准的Markdown语法编写，该写标题的写标题，该用列表的用列表，该用表格的用表格，该用代码块的标明puppet/bash语法高亮，会严格遵守一本技术书籍的基本修养，因此就不再废话。

Module命名规范说明

绝大多数的puppet modules命名规范使用puppet-前缀，后面为服务名称，例如：

- puppet-nova
- puppet-keystone

当然也有一些公共库模块，例如:

- puppet-oslo
- puppet-openstacklib

术语表

本书的后续章节中使用了与Puppet，Ruby，OpenStack等技术相关的术语，为了让读者更快地理解其含义，下表中给出了本书所出现的术语和释义。

名称	说明
facter	用于获取系统变量的组件
puppet	当出现在终端时，表示puppet软件的命令行工具；当出现在文中时，可能表示puppet软件，或者puppet client端
resource	puppet中的资源单位，你可以认为它与“Linux中一切皆file“这句话对等
class	puppet中resource的集合，与面向对象中的类无关
define	puppet中resource的集合，与编程语言中的函数定义无关
module	puppet中class和define的集合，与服务紧密相关，例如puppet-apache，专门管理apache所有相关配置
transformation layer	转换层，可以理解为对class和define的调用层
manifests	用于puppet代码的文件目录
node definition	节点定义文件，等价于角色定义
hiera	数据文件，用于存放节点所有变量的赋值
RVM	安装和管理多个Ruby环境以及Ruby应用所使用的Ruby环境。
Rails	Web开发框架
RubyGems	RubyGems是一个方便而强大的Ruby程序包管理器（package manager），类似RedHat的RPM.它将一个Ruby应用程序打包到一个gem里，作为一个安装单元。无需安装，最新的Ruby版本已经包含RubyGems了
Gem	Gem是封装起来的Ruby应用程序或代码库。
Gemfile	定义你的应用依赖哪些第三方包，bundle根据该配置去寻找这些包。
Rake	Rake是一门构建语言，和make类似。Rake是用Ruby写的，它支持自己的DSL用来处理和维护Ruby程序。 Rails用rake扩展来完成多种不容任务，如数据库初始化、更新等。 详细 http://rake.rubyforge.org/
Rakefile	Rakefile是由Ruby编写，Rake的命令执行就是由Rakefile文件定义。
Bundle	Bundler为Ruby应用维护了一个持久的包依赖环境。

- 术语表

PuppetOpenstack项目简介

`PuppetOpenstack` 项目是由PuppetLabs公司于2013年发起的开源项目，最初托管在Github上，半年后移入OpenStack CI体系。

PuppetOpenstack最初只有Keystone,Nova,Glance,Cinder等几个核心项目的modules，随后得到了Red Hat,Cisco,Mirantis等公司的广泛支持，在社区贡献者的持续努力下，PuppetOpenstack项目从Stackforge孵化项目演变成了OpenStack Offical项目，目前隶属于Openstack Goverance项目。目前已构成了了一套庞大而复杂的部署体系。

你可以通过以下链接找到有关PuppetOpenstack项目的更多细节说明：

- Wiki（Out of date）: https://wiki.openstack.org/wiki/Puppet
- Docs: http://docs.openstack.org/developer/puppet-openstack-guide/

为什么选择PuppetOpenstack

PuppetOpenstack社区对与其目标的定义如下：

> to bring scalable and reliable IT automation to OpenStack cloud deployments.

目前用于部署OpenStack的工具已非常广泛，为什么要选择它呢？或者说从技术角度来看，OpenStack自动化部署工具应该如何做技术选型？

笔者认为有以下三点：

第一，PuppetOpenstack项目诞生于2013年，诞生时间早，参与贡献者众多，使得PuppetOpenstack项目非常成熟和稳定，这对于自动化运维来说是十分重要的考虑因素。

其次，2016年4月出炉的Openstack User Survey (https://www.openstack.org/user-survey/survey-2016-q1/landing)

下图选自该份报告，统计了关于主流部署工具种类和占有率的使用统计：

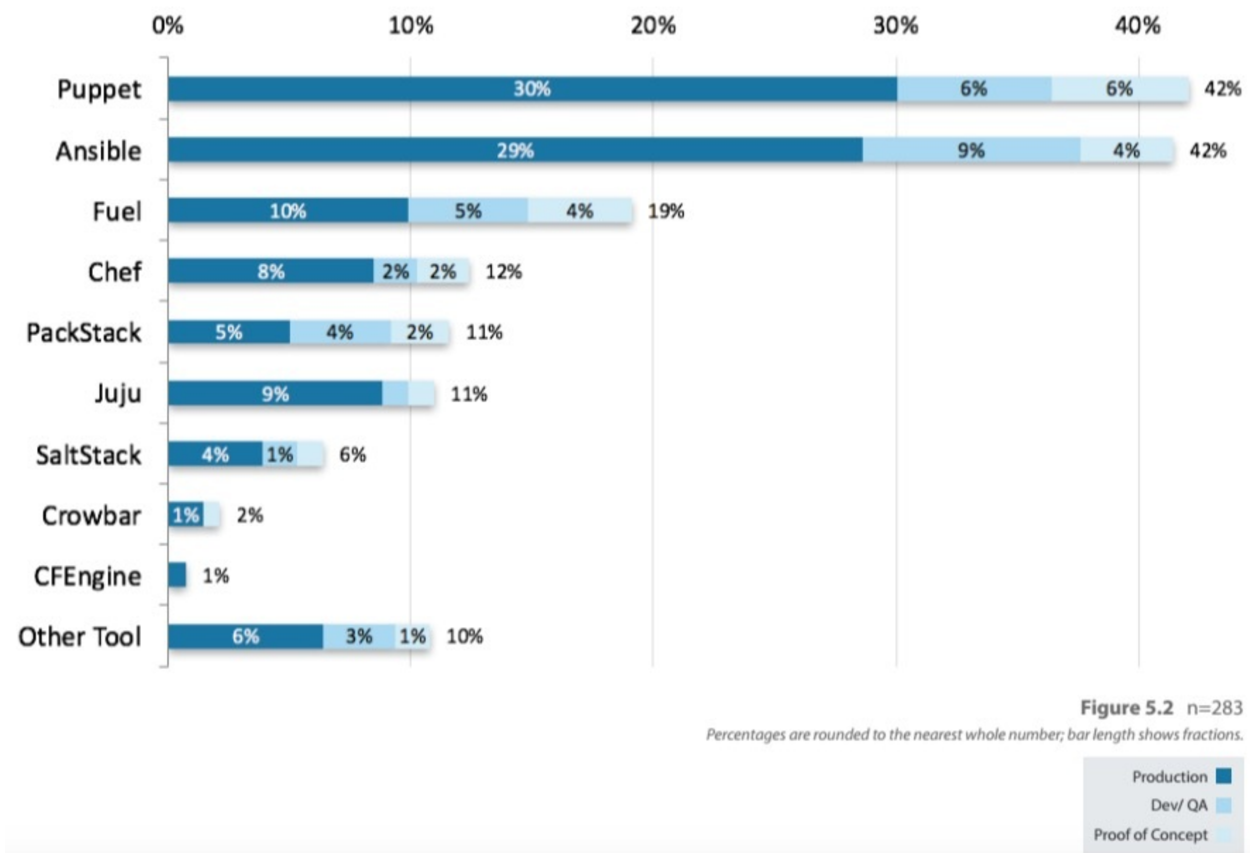


Figure 5.2 n=283

Percentages are rounded to the nearest whole number; bar length shows fractions.

值得一提的是，图中Fuel和Packstack项目的核心部署功能直接使用的是PuppetOpenstack项目。因此，可以理解为有近乎一半的用户选择使用PuppetOpenstack部署Openstack，这对于百花齐放的开源世界来说，是非常可观而且有说服力的数字。

第三，欧洲原子能机构CERN使用Puppet管理着世界上规模最为庞大的OpenStack集群，总计超过了一万台服务器。这从用户角度证明了PuppetOpenstack可以支持大规模Openstack集群的部署。

OpenStack modules

现在我们先站在最高的山峰上，来看看这些伟岸的群山吧。

第一个印入我们眼帘的是Openstack服务相关的modules，目前puppetopenstack已经支持以下服务的配置和管理：

- Alarming (Aodh)
- Key Manager (Barbican)
- Telemetry (Ceilometer)
- Block Storage (Cinder)

- DNS (Designate)
- Image service (Glance)
- Time Series Database (Gnocchi)
- Orchestration (Heat)
- Dashboard (Horizon)
- Bare Metal (Ironic)
- Identity (Keystone)
- Shared Filesystems (Manila)
- Workflow service (Mistral)
- Application catalog (Murano)
- Networking (Neutron)
- Compute (Nova)
- Load Balancer (Octavia)
- Oslo libraries (Oslo)
- Benchmarking (Rally)
- Data processing (Sahara)
- Object Storage (Swift)
- Testing (Tempest)
- Deployment (TripleO)
- Database service (Trove)
- Deployment UI (TripleO UI)
- Root Cause Analysis (Vitrage)
- Message service (Zaqar)

Tool modules

第二大山脉是工具类相关的modules，分别有：

- Common Puppet library (OpenStackLib)
- Common Ruby helper library (puppet-openstack_spec_helper)
- Puppet OpenStack helpers (OpenStackExtras)
- Virtual Bridging (OpenvSwitch)
- Integration CI tools (Puppet OpenStack Integration)
- Blueprints (Puppet OpenStack Specs) (hosted here)
- Compliant tool (Cookiebutter)
- Sync tool (Modulesync)

Other modules

第三大块则是一些尚在开发阶段或者已经废弃的模块：

- Storage (Ceph)
- Monitoring (Monasca)
- Composition Layer (deprecated in Juno) (OpenStack)

推荐的阅读顺序

> 说明： 本书以**Ocata**版本为基础

如果你是第一次接触PuppetOpenstack，推荐从 `公共库和工具类模块` 章节的 `puppet-openstack-integration` 一节开始，这节会介绍如何使用PuppetOpenstack模块快速部署一个All-in-One的Openstack服务。

第二章 Puppet开发基础

在开始部署OpenStack前，首先你要完成一些准备工作，因为OpenStack是一个复杂的技术栈组合，部署工作同样不简单，在你准备进入每个特定章节前，我们已为你准备好了相应的知识。

环境搭建

你可以使用我们准备的安装脚本来配置实验环境，请在终端下执行以下命令:

```
sudo curl http://pom.nops.cloud/scripts/install_example_environment
```

建议为你的虚拟机制作好快照，在调试代码时，尽可能使用纯净的测试环境。

其他准备

- 请系好安全带，我们准备起飞了 :D

关于Puppet

Puppet简介

Puppet是由Puppet公司开发的系统管理框架和工具集，被用于IT服务的自动化管理。由于良好的声明式语言和易于扩展的框架设计以及可重用可共享的模块，使得Google、Cisco、Twitter、RedHat、New York Stock Exchange等众多公司和机构在其数据中心的自动化管理中用到了Puppet。

因为得到众多开发者和用户的支持，Puppet长期保持着IT自动化管理领域的领头羊。

为什么Puppet能得到青睐？笔者认为有以下几点：

- 功能强大的DSL，较为平滑的学习曲线
- 良好的扩展性
- 可被复用和共享的模块
- 活跃的社区
- 详细的文档

Puppet公司简介

Puppet是由Puppet公司所有，曾用名为Puppetlabs。

除了自家的拳头产品Puppet之外，Puppet公司还提供了一些相关领域的运维自动化工具。

除了软件，Puppet公司还有两项值得关注的技术输出：

- 半年一度的PuppetConf大会已跻身于IT圈的顶级技术会议之列
- 每年发布的State of DevOps Report是DevOps领域最具影响力的调查报告

基础知识要求

- 对Linux基础知识有所了解，推荐《鸟哥的Linux私房菜 基础学习篇》
- 对Puppet基础知识有所了解 推荐 官方学习文档

- 对OpenStack部署有所了解

同类工具

除了Puppet之外，在业界常见的配置管理工具还有：

- CFengine 老牌配置管理工具
- Chef 和Puppet相似的配置管理工具
- Saltstack 使用Python编写的配置管理和编排工具
- Ansible 使用Python编写的配置管理和编排工具

Puppet核心概念

本书的重点是讲解PuppetOpenstack项目，并假定读者对于Puppet有一定的了解，因此将不会包含对于Puppet基础知识的讲解。

然而，在Puppet中有一些非常重要的概念，对于这些核心概念的准确理解，将有助于读者快速掌握Puppet Modules的开发，因此，本节将花费一些篇幅来帮助读者深入理解这些核心概念。

0.Resource Type

在Linux中，一切皆文件(`file`)。而在Puppet中，一切皆资源(`resource`)。

比如，package对应着 `软件包` 资源类型(resource type)。

在服务器上安装vim软件包，相应地声明一个package资源:

```
package {'vim':
  ensure => present
}
```

在服务器上管理ntp服务，相应地声明一个service资源：

```
service {'ntpd':
  ensure => running
}
```

在Puppet中，常用的资源类型有以下8类：

- file
- package
- service
- modify
- exec
- cron
- user

- group

0.1 资源声明

资源声明（Resource declaration）则是一个表达式，用于描述资源的期望状态并将其添加到Catalog。

可以理解为类似于编程语言中的函数调用。

1.Class

与面向对象语言不同，类(Class)在Puppet中只是表示了一个代码块：通常将一些相关的功能组合到一起，并存储到module中，以便后期使用。

1.1 类的定义

定义一个Class的语法格式如下：

- 以 `class` 关键字开头
- 指定一个类的名称
- 参数列表（可选）
- 一对花括号
- 至少含有一个资源声明的代码块

例如，以下是一个关于apache的Class：

```
class apache (String $version = 'latest') {
  package {'httpd':
    ensure => $version, # Using the class parameter from above
    before => File['/etc/httpd.conf'],
  }
  file {'/etc/httpd.conf':
    ensure  => file,
    owner   => 'httpd',
    content => template('apache/httpd.conf.erb'), # Template from a
  }
  service {'httpd':
    ensure    => running,
    enable    => true,
    subscribe => File['/etc/httpd.conf'],
  }
}
```

1.2 Class文件的存放位置

Class定义文件应存放在modules的manifests目录下，Puppet将自动地加载该路径下的所有类。

1.3 类的声明

在Puppet manifest文件中声明一个Class时，则会将其添加到catalog文件。通常在节点定义文件中或者其他class文件中去声明一个Class。

在Puppet中，有两种方式来声明一个Class：

- 类Include方式
- 类Resource声明方式

1.3.1 Include方式

Include方式是指使用 `include` ， `require` ， `contain` ， `hiera_include` 函数来声明Class，使用这种方式Class可以安全地被多次声明。

例如：

```
class compute(){
  include ::nova
  include ::nova::api
}

node 'compute_node' {
  include compute
  # nova类被声明了2次
  include ::nova
}
```

何谓安全地多次声明？

> 任何一个Class在一个指定节点的定义中，只能被声明一次，否则Puppet在运行时会抛出资源重复声明的错误。这也是初学者容易犯错的地方。

而通过类include的方式可以实现尽管Class被多次声明，但最终只向catalog添加一次的效果。

但使用这种方式，则Class中的参数传值只能通过Hiera进行。

1.3.2 Class方式

Class的方式则要求每个被声明的Class只被声明一次。通过这种方式，在声明某个特定Class的时候，可以对指定参数进行重新赋值。

例如：

```
class compute($ip='127.0.0.1'){
  class {'nova':
    ipaddress => $ip
  }
  class {'nova::api':}
}

node 'compute_node' {
  class{'compute':
    ip => '192.168.1.1'
  }
}
```

2.Defines

Defines也称为是Defined resource type，是一段可以被多次赋值的代码块，可以理解为是一种轻量级的自定义的资源类型。

例如，以下是nova::manage:network define，用于管理nova network的创建。在实际使用中，可以通过传递不同的参数给nova::manage::network来创建不同的nova network。

```
define nova::manage::network (
  $network,
  $label          = 'novanetwork',
  $num_networks   = 1,
  $network_size   = 255,
  $vlan_start     = undef,
  $project        = undef,
  $allowed_start = undef,
  $allowed_end    = undef,
  $dns1           = undef,
  $dns2           = undef
) {

  include ::nova::deps

  nova_network { $name:
    ensure        => present,
    network       => $network,
    label         => $label,
    num_networks  => $num_networks,
    network_size  => $network_size,
    project       => $project,
    vlan_start    => $vlan_start,
    allowed_start => $allowed_start,
    allowed_end   => $allowed_end,
    dns1          => $dns1,
    dns2          => $dns2,
  }

}
```

初学者在选择如何define和class时，常常犹豫不决。

首先，来看这两种类型的最大区别：

- Define：在一个catalog中可以被重复声明
- Class： 在一个catatlog中只能被声明一次

再谈使用场景：

- Class通常用于管理具有唯一性的资源
- Define通常用于管理具有多样性的资源

 以Apache为例，会使用Class来管理Apache软件包，主配置文件，以及服务状态的管理；而Apache vhost则会使用Define来管理。我们会在后面 `puppet-apache` 章节中详细讲解。

3. Nodes

假设你已经下载了puppet-apache和puppet-mysql模块，接下来要为指定服务器赋予指定的角色，那么这个过程称为是节点分类（Node Classification）。在Puppet中，这些数据通常存储在节点定义文件中。

节点定义文件的存放路径通常位于 `<ENVIRONMENTS DIRECTORY>/<ENVIRONMENT>/manifests/site.pp` 。

现在我们要配置2种类型的节点：Web服务器 `www.example.com` 和DB服务器 `db1.example.com` ，在site.pp中加入以下代码：

```
node 'www.example.com' {
  include apache
}
node 'db.example.com' {
  include mysql
}
```

最佳实践

尽管在节点定义文件里可以添加任何的Puppet代码，但请保持只在节点定义文件中做两件事情：声明类和设置变量。

理解Hiera

0. Hiera是什么？

Hiera是Puppet内建的键值类型数据查询系统。Hiera默认支持的存储后端格式是YAML和JSON，当然也可以根据需要编写自定义的后端接口。

1. Hiera的历史和意义

在Puppet 2.x的早期版本中，在声明一个节点的角色时，与该节点相关的数据是直接与类的声明相关联。这种混合的方式，随着被管理集群数量的增加而变得复杂不堪。因此，社区提出了一个称之为Hiera的额外组件，可以将节点定义和节点数据分离开，但在2.x中，想要使用Hiera函数，则须安装额外的软件包，并且对已有的代码进行修改。从Puppet 3.x起，Hiera作为Puppet的原生功能，可以在manifests中直接使用。

通过描述比较晦涩，下面来看一个实际的例子：

假设我们在节点定义文件中声明了100台Web服务器，其中前三台Web服务器的定义如下：

```
node 'web01' {
  include apache
}

node 'web02' {
  class {'apache':
    default_vhost => false
  }
}

node 'web03' {
  class {'apache':
    default_mods => true
  }
}
```

从上述代码，可以发现每个节点都声明了 `class apache` ，有的使用了apache类的默认值，有的则对某些参数传递了非默认值。

如果这100台Web服务器节点定义的赋值都不相同，那么这个节点定义文件代码的行数将长达几百行，对于管理员来说则是一个梦靥。

在引入Hiera之后，我们可以看到明显的不同：

1.节点定义文件

```
# 使用正则来匹配所有的web角色节点
node /^web\d+$/ {
  include apache
}
```

2.节点数据文件

web02.yaml

```
---
# 变量使用类名标识的命名空间来区分
apache::default_vhost: false
```

web03.yaml

```
---

apache::default_mods: true
```

在上述yaml文件中，参数 `$defualt_vhost` 与之对应的 `apache::default_vhost` 称为Hiera key， `false` 是key value。

通过这种方式，可以做到将所谓的节点定义和节点数据相互分离，减少了冗余代码，提高了代码可读性，也提高了安全性。

2.Hiera配置文件

Hiera的配置文件称为hiera.yaml。在Hiera 4之前，hiera.yaml是全局唯一的，在Hiera 5之后，hiera.yaml则被分成了三类：全局，环境相关，模块相关。但是其目的都是一样的：定义数据的层次。

在解释层次这个概念之前，先理解在前一个Web节点例子中的Hiera配置文件是如何编写的。

以下是环境相关的hiera配置文件：

```
# /hiera.yaml
---
version: 5

defaults:  # Used for any hierarchy level that omits these keys.
  datadir: data  # This path is relative to the environment -- /dat
  data_hash: yaml_data  # Use the built-in YAML backend.

hierarchy:
  - name: "Per-node data"                    # 可读的名称
    path: "nodes/%{trusted.certname}.yaml"  # 文件路径
```

继续以Web节点为例，现在需要在Web节点中加入对NTP服务的管理：

```
node /^web\d+$/ {
  include apache
  include ntp
}
```

在 `class ntp` 有一个参数是 `$ntp_server` ，需要在每一个web节点的yaml文件中添加以下参数：

```
---
ntp::ntp_server:
 - '10.0.10.101'
 - '10.0.10.102'
 - '10.0.10.103'
```

如果有100个节点，需要重复添加100次。

那么如何消除这段冗余的数据呢？还记得我们先前提到的层次（hierarchy）吗？

接下来，我们在现有的hiera.yaml添加一个新“层次”：common.yaml,用于统一管理web节点的公共数据。

```
# /hiera.yaml
---
version: 5

defaults:  # Used for any hierarchy level that omits these keys.
  datadir: data  # This path is relative to the environment -- /dat
  data_hash: yaml_data  # Use the built-in YAML backend.

hierarchy:
  - name: "Per node data"                      # 可读的名称
    path: "nodes/%{trusted.certname}.yaml"  # 文件路径

  - name: "Common data"
    path: "common.yaml"
```

在common.yaml中对$ntp_server赋值:

```
---
ntp::ntp_server:
 - '10.0.10.101'
 - '10.0.10.102'
 - '10.0.10.103'
```

通过Hiera的层次设定，不同的节点通过同一套Hiera可以得到不同的数据，实现了数据的复用和分离等运维目标。

3.小结

Hiera是Puppet中的核心组件，也属于不易理解的部分。本节通过一些简单的示例来说明Hiera的主要功能，而Hiera还拥有其他强大的功能，推荐读者深入阅读以下官方文档：

- Hiera: How hierarchies work
- Hiera: How the three config layers work
- Hiera: Merging data from multiple sources

- 理解Hiera

准备开发测试环境

0.环境准备

在开始介绍PuppetOpenstack前，我们需要准备一台虚拟服务器用于接下来的练习。

读者可以通过使用虚拟化软件或者通过云平台创建一台虚拟机。

其规格如下：

- `2 vCPU`, `4G RAM`, `30G Disk`,至少有一块`NIC`，操作系统为`CentOS 7.1/7.2`，可以访问Internet

在安装Puppet之前，需要为虚拟主机设置合适的主机名，域名，时间等。

```
$ hostnamectl set-hostname learnpom

$ echo "127.0.1.1 learnpom.example.in learnpom" >> /etc/hosts
```

1.了解Puppet

在安装Puppet前，首先需要了解Puppet的运行方式，当前Puppet支持两种运行方式：

- Server/Client模式，需要安装Puppet agent和Puppet server软件包
- Standalone模式，只需要安装Puppet agent软件包

在通常的开发场景下，笔者推荐使用Standalone模式，操作简单，定位问题容易；在管理内部的测试/生产环境时，笔者建议须使用Server/Client模式，进行集中式管理。

本书中除个别场景外，默认以Standalone模式为主。

2.安装Puppet

Puppet由三个软件包构成：

- puppet-agent: 用于安装Puppet,Ruby,Facter,Hiera和依赖包的软件包
- puppetserver: 用于安装Puppet Server服务

> 注：本文所使用的Puppet版本是4.1x

打开虚拟机的终端，使用root权限在命令行下输入以下命令：

```
$ cat << EOF >> install_puppet.sh

# Script for installing puppet Based on CentOS 7.x

set -e

if [ -n "$DEBUG" ]; then
  set -x
fi

# set environment
export SCRIPT_DIR=$(cd `dirname $0` && pwd -P)
export PUPPET_VERSION=${PUPPET_VERSION:-4}
export MANAGE_PUPPET_MODULES=${MANAGE_PUPPET_MODULES:-true}
export MANAGE_REPOS=${MANAGE_REPOS:-true}
export PUPPET_ARGS=${PUPPET_ARGS:-}
export SCRIPT_DIR=$(cd `dirname $0` && pwd -P)
if [ $PUPPET_VERSION == 4 ]; then
  export PATH=${PATH}:/opt/puppetlabs/bin
  export PUPPET_RELEASE_FILE=puppetlabs-release-pc1
  export PUPPET_BASE_PATH=/etc/puppetlabs/code
  export PUPPET_PKG=puppet-agent
elif [ $PUPPET_MAJ_VERSION == 5 ]; then
  export PATH=${PATH}:/opt/puppetlabs/bin:/opt/puppetlabs/puppet/bi
  export PUPPET_RELEASE_FILE=puppet5-nightly-release
  export PUPPET_BASE_PATH=/etc/puppetlabs/code
  export PUPPET_PKG=${PUPPET_PKG:-puppet-agent}
fi
if [ $(id -u) != 0 ]; then
  # preserve environment so we can have ZUUL_* params
  SUDO='sudo -E'
```

```
fi

echo 'Setup (RedHat based)'
sudo yum -y remove facter puppet rdo-release
sudo yum -y install libxml2-devel libxslt-devel ruby-devel rubygems
sudo yum -y groupinstall "Development Tools"

echo 'Install Bundler'
mkdir -p .bundled_gems
export GEM_HOME=`pwd`/.bundled_gems
gem install bundler --no-rdoc --no-ri --verbose

echo 'Start install puppet'

if rpm --quiet -q $PUPPET_RELEASE_FILE; then
    $SUDO rpm -e $PUPPET_RELEASE_FILE
fi
# EPEL does not work fine with RDO, we need to make sure EPEL is re
if rpm --quiet -q epel-release; then
    $SUDO rpm -e epel-release
fi
$SUDO rm -f /tmp/puppet.rpm

wget  http://yum.puppetlabs.com/${PUPPET_RELEASE_FILE}-el-7.noarch.
$SUDO rpm -ivh /tmp/puppet.rpm
$SUDO yum install -y dstat ${PUPPET_PKG} setools setroubleshoot auc
$SUDO service auditd start

# SElinux in permissive mode so later we can catch alerts
$SUDO setenforce 0
EOF

$ sudo bash install_puppet.sh
```

3.安装PuppetServer

Puppetserver的手动安装和配置部署比较繁杂，但是Puppet的目标不就是实现软件安装部署的自动化吗？

因此，我们可以使用 `puppet module` 安装用于部署Puppet Server的module，然后完成Puppetserver的一键安装。

在终端下执行以下命令：

```
$ puppet module install theforeman-puppet

$ cat > install.pp < true, server_foreman => false }
EOF

$ puppet apply install.pp -v
```

第二章 OpenStack基础服务模块

本章概览

何为基础模块？

在OpenStack生产环境中，对外提供API服务的组件运行在Web服务器上，服务组件之间大多通过消息队列（MQ)进行通讯，认证使用的Token数据则会被存到缓存服务中，租户的虚拟机，块设备等等资源信息均被保存到关系型数据库中等等场景，都用到了大量的基础服务。那么在本章将会介绍如何使用Puppet来管理这些基础服务。

在Puppet中，与基础模块对应的是：

- OpenStack使用到的公共基础服务，如数据库，消息队列，缓存，Web服务器等等
- 与操作系统相关的配置模块，如防火墙，网络配置等。

每一小节会介绍一个单独的Puppet模块，每个模块的内容统一地划分为1-6个部分：

- 基础知识　　你在讲什么？给我讲讲基础先。
- 先睹为快　　拜托！先别说那些无聊的理论和代码剖析和说教，run起来让我看看效果先。
- 核心代码　　还有这种操作？老司机，带带我。
- 使用说明　　常见和经典的使用用例
- 小结　　　　我们刚才都讲了什么？
- 课后练习　　嗯，我感觉我什么都明白了。是吗？来，你来握方向盘。

在每一节内容里，会穿插一些重要function,resource type,facter的使用技巧，会涉及到一些理论知识，再扯一点历史。

例如，在 `puppet-oslo` 模块章节，会讲到为什么Puppet原生不支持迭代，如何去实现。当然，为了避免偏离本书主旨，笔者会点到为止。如果你对这些分支知识感兴趣的话，可以跳转到本书给出的参考链接，继续深陷其中。

puppet-apache

1. 先睹为快
2. 代码讲解—如何管理apache服务
3. 推荐阅读
4. 动手练习

Apache HTTP Server（简称Apache）是Apache软件基金会的一个开放源代码的网页服务器软件，可以在大多数电脑操作系统中运行，由于其跨平台和安全性被广泛使用，是最流行的Web服务器软件之一。它快速、可靠并且可通过简单的API扩充，将Perl／Python等解释器编译到服务器中。

`puppet-apache` 模块是由Puppet公司维护的官方模块，提供了完善的Apache管理能力。

`puppet-apache` 项目地址：https://github.com/puppetlabs/puppetlabs-apache

在开始介 `puppet-apache` 模块前，读者需特别留意以下：

> WARNING: Configurations not managed by Puppet will be purged.

对于已存在的Apache服务，如果尝试使用 `puppet-apache` 模块进行管理，请额外小心在默认情况下该模块会清除所有未被Puppet管理的配置文件！

1.先睹为快

不想看下面大段的代码说明，已经跃跃欲试了？

Ok，我们开始吧！

打开虚拟机终端并输入以下命令：

```
$ puppet apply -ve "include ::apache"
```

或者创建一个manifest文件test.pp，并输入以下代码：

```
class { 'apache': }
```

在终端下执行 `puppet apply` 命令:

```
puppet apply -v test.pp
```

约1分钟之后（取决于网速和虚拟机的性能），Puppet已经完成了Apache服务的安装，配置和启动了。

这是如何做到的呢？我们打开puppet-apache模块下manifests/init.pp文件，看看是如何做的？

2.代码讲解

`puppet-apache` 模块当前支持的主要功能如下:

- Apache配置文件和目录
- Apache软件包/服务/配置文件
- Apache的module
- 虚拟主机(Virtual hosts)
- 监听端口(Listened-to ports)

2.1 `class apache`

`class apache` 中有大量的判断逻辑，这些并不是核心，对于一个init类，其核心是调用了哪些资源（class，define等）：

用于安装Apache软件包

```
package { 'httpd':
  ensure => $package_ensure,
  name   => $apache_name,
  notify => Class['Apache::Service'],
}
```

用于管理conf.d目录，注意这个$purge_confd参数，默认为true,会清理掉一切未被管理的配置文件。

```
  file { $confd_dir:
    ensure  => directory,
    recurse => true,
    purge   => $purge_confd,
    notify  => Class['Apache::Service'],
    require => Package['httpd'],
  }
```

用于启用所有默认的mods

```
class { '::apache::default_mods':
  all => $default_mods,
}
```

这里有两个apache::vhost define，分别用于生成默认的80端口和443端口的vhost文件。

```
    ::apache::vhost { 'default':
      ensure          => $default_vhost_ensure,
      port            => 80,
      docroot         => $docroot,
      scriptalias     => $scriptalias,
      serveradmin     => $serveradmin,
      access_log_file => $access_log_file,
      priority        => '15',
      ip              => $ip,
      logroot_mode    => $logroot_mode,
      manage_docroot  => $default_vhost,
    }
    $ssl_access_log_file = $::osfamily ? {
      'freebsd' => $access_log_file,
      default   => "ssl_${access_log_file}",
    }
    ::apache::vhost { 'default-ssl':
      ensure          => $default_ssl_vhost_ensure,
      port            => 443,
      ssl             => true,
      docroot         => $docroot,
      scriptalias     => $scriptalias,
      serveradmin     => $serveradmin,
      access_log_file => $ssl_access_log_file,
      priority        => '15',
      ip              => $ip,
      logroot_mode    => $logroot_mode,
      manage_docroot  => $default_ssl_vhost,
    }
  }
```

以上代码示例用于简单的测试验证，若在生产环境中，请关闭默认生成的vhost文件：

```
class { 'apache':
  default_vhost => false,
}
```

2.2 配置Apache mod

puppet-apache 支持使用两种方式来安装mod软件包和管理mod配置文件：

- class apache::mod::<MODEULE_NAME> 方式
- define apache::mod 方式

其中 apache::mod::<MODULE NAME> 支持众多已预先定义的Apache mod的管理，而 define apache::mod 方式则可以灵活地支持未在 define apache::mod 中的mod。

2.2.1 class apache::mod::ssl

下面以 mod_ssl 为例进行说明：

为了确保通讯安全，会使用HTTPS来加密通讯，因此需在Apache启用mod_ssl。

- class apache::mod::<MODEULE_NAME> 方式：

```
#开启ssl compression
class { 'apache::mod::ssl':
  ssl_compression => true,
}
```

- define apache::mod 方式：

  ```
  apache::mod { 'mod_ssl': }
  ```

在通常情况下，使用默认参数apache::mod::ssl就可以完成mod_ssl的管理工作，同时也提供了10个可配置的参数：

- $ssl_compression = false
- $ssl_cryptodevice = 'builtin'
- $ssl_options = ['StdEnvVars']
- $ssl_openssl_conf_cmd = undef
- $ssl_cipher = 'HIGH:MEDIUM:!aNULL:!MD5:!RC4'
- $ssl_honorcipherorder = 'On'
- $ssl_protocol = ['all', '-SSLv2', '-SSLv3']

- $ssl_pass_phrase_dialog = 'builtin'
- $ssl_random_seed_bytes = '512'
- $ssl_sessioncachetimeout = '300'

需要注意的是，在使用 `define apache::mod` 方式下，Puppet仅会安装指定名称的mod软件包，用户需要手动完成对于mod配置文件的设置。

2.2.2 `class apache::mod::wsgi`

OpenStack服务的所有提供API接口的组件使用Python语言编写，Python原生的Web服务器性能较弱，只适合用于非线上环境。为了提高API服务的性能，需将Python Web程序运行在Apache上，将使用到 `mod_wsgi` 。

在通常情况下,声明 `apache::mod::wsgi` 时使用默认参数就可以完成 `mod_wsgi` 的安装和配置工作：

```
class { 'apache::mod::wsgi':}
```

`apache::mod::wsgi` 也提供了5个可配置的参数，其中 `$wsgi_socket_prefix` 有默认值：

- `$wsgi_socket_prefix` = $::apache::params::wsgi_socket_prefix
- `$wsgi_python_path`
- `$wsgi_python_home`
- `$package_name`
- `$mod_path`

2.3 `define apache::vhost`

在配置Apache时，最常见的操作之一就是添加和修改虚拟主机。

因此，在 `puppet-apache` 模块中 `apache::vhost` 是使用最频繁的define，用于管理Apache服务的vhost配置文件。

2.3.1 配置一个vhost

最简单的调用方式是在声明一个 `apache::vhost` 时，只对参数port和docroot传值，例如：

```
apache::vhost { 'vhost.example.com':
  port    => '80',
  docroot => '/var/www/vhost',
}
```

2.3.2 配置开启SSL的vhost

在线上配置vhost时，经常会使用HTTPS来确保Web访问的安全性，这在puppet中配置起来也非常容易。在声明一个 `apache::vhost` 时，开启$ssl参数即可：

```
apache::vhost { 'ssl.example.com':
  port    => '443',
  docroot => '/var/www/ssl',
  ssl     => true,
}
```

如果要对开启SSL的vhost指定证书路径，则在声明时引入参数 `ssl_cert` 和 `ssl_key` ：

```
apache::vhost { 'cert.example.com':
  port     => '443',
  docroot  => '/var/www/cert',
  ssl      => true,
  ssl_cert => '/etc/ssl/cert.example.com.cert',
  ssl_key  => '/etc/ssl/cert.example.com.key',
}
```

2.3.3 配置一个WSGI的vhost

下面代码示例说明了如何为vhost配置WSGI mod，用于运行Python Web服务：

```
apache::vhost { 'wsgi.example.com':
  port                        => '80',
  docroot                     => '/var/www/pythonapp',
  wsgi_application_group      => '%{GLOBAL}',
  wsgi_daemon_process         => 'wsgi',
  wsgi_daemon_process_options => {
    processes    => '4',
    threads      => '24',
    display-name => '%{GROUP}',
  },
  wsgi_import_script          => '/var/www/wsgi.example.com',
  wsgi_import_script_options  => {
    process-group     => 'wsgi',
    application-group => '%{GLOBAL}',
  },
  wsgi_process_group          => 'wsgi',
  wsgi_script_aliases         => { '/' => '/var/www/wsgi.example.co
}
```

推荐阅读

- ServerLimit
- ServerName
- ServerRoot
- ServerTokens
- ServerSignature
- Service attribute restart
- mod_wsgi
- mod_ssl

动手练习

1. 使用Puppet搭建一套LAMP环境（注：需和 `puppet-mysql` 结合使用）
2. 使用 `Certbot` 和 `puppet-apache` 配置并管理一个HTTPS站点 (注： `Certbot` 的说明见https://certbot.eff.org/)

`puppet-memcached` 模块

Memcached是一个高性能的分布式内存对象缓存系统，用于动态Web应用以减轻数据库负载，最初由LiveJournal的Brad Fitzpatrick开发，目前得到了广泛的使用。它通过在内存中缓存数据和对象来减少读取数据库的次数，从而提高动态、数据库驱动网站的速度。

`puppet-memcached` 是由Steffen Zieger(saz)维护的一个模块。同时，他还维护了 `puppet-sudo` , `puppet-ssh` 等模块。

`puppet-memcached` 项目地址：https://github.com/saz/puppet-memcached

1.先睹为快

不想看下面大段的代码解析，已经跃跃欲试了？

OK，我们开始吧！

打开虚拟机终端并输入以下命令：

```
$ puppet apply -e "class { 'memcached': }"
```

在看到赏心悦目的绿字后，Puppet已经完成了Memcached服务的安装，配置和启动。这是如何做到的呢？

我们打开 `puppet-memcached` 模块下 `manifests/init.pp` 文件来一探究竟吧。

2.代码解析

`puppet-memcached` 模块的代码结构非常简洁，所有的工作都在 `Class memcached` 中完成：

2.1 `Class memcached`

1.以下代码完成了对 `Memcached` 软件包管理：

```
package { $memcached::params::package_name:
  ensure   => $package_ensure,
  provider => $memcached::params::package_provider
}

if $install_dev {
  package { $memcached::params::dev_package_name:
    ensure  => $package_ensure,
    require => Package[$memcached::params::package_name]
  }
}
```

此处，值得一提的是：在 `package` 资源类型中，需要重写参数 `provider` 默认值的情况并不常见，该参数用于设置管理软件包的后端，常见的可选项有： `yum` , `apt` , `pip` 等。

2.下述代码完成了对 `Memcached` 服务的管理：

```
if $service_manage {
  service { $memcached::params::service_name:
    ensure     => $service_ensure,
    enable     => $service_enable,
    hasrestart => true,
    hasstatus  => $memcached::params::service_hasstatus,
  }
}
```

在 `service` 资源类型中，需要设置参数 `hasstatus` 的情况也并不多见，该参数用于设置目标服务是否具有查看服务状态的脚本，默认为 `true` 。如果该服务的软件包中并没有提供查看服务运行状态的脚本，可以添加 `status` 参数，来用于指定一个手动运行的命令：若返回值为0，则认为服务是运行状态；若返回值非0，则认为服务是非运行状态。

2.2 `memcached_sysconfig.erb` 模板

在 `class memcached` 中使用了 `file` 资源对 `Memcached` 配置文件进行管理：

```
  if ( $memcached::params::config_file ) {
    file { $memcached::params::config_file:
      owner   => 'root',
      group   => 'root',
      mode    => '0644',
      content => template($memcached::params::config_tmpl),
      require => Package[$memcached::params::package_name],
      notify  => $service_notify_real,
    }
  }
```

2.2.1 什么是模板

第一次见到了模板(`template`)，这是Puppet用于管理配置文件的常用方法。

模板是指含有可执行代码和数据的特殊文本格式文件，通过渲染最终生成纯文本文件。使用模板的目标就是通过一些简单的输入（传递几个参数）就可以产生复杂的文本输出。

在 `memcached::params` 中查询到RHEL下的 `$memcached::params::config_tmpl` 值为 `${module_name}/memcached_sysconfig.erb` 。

`.erb` 又称为 `Embedded Ruby` 模板语言，Puppet可以通过函数 `template` 和 `inline_template` 来渲染模板文件。

下面取自 `templates/memcached_sysconfig.erb` 文件的部分代码片段。

```
<%-
result = []
if @verbosity
  result << '-' + @verbosity.to_s
end

...

if @extended_opts
  result << '-o ' + @extended_opts.join(',')
end
result << '-t ' + @processorcount.to_s

# log to syslog via logger
if @syslog && @logfile.empty?
    result << '2>&1 |/bin/logger &'
# log to log file
elsif !@logfile.empty? && !@syslog
  result << '>> ' + @logfile + ' 2>&1'
end
-%>
<%- if scope['osfamily'] != 'Suse' -%>
PORT="<%= @tcp_port %>"
USER="<%= @user %>"
MAXCONN="<%= @max_connections %>"
<% Puppet::Parser::Functions.function('memcached_max_memory') -%>
CACHESIZE="<%= scope.function_memcached_max_memory([@max_memory]) %
OPTIONS="<%= result.join(' ') %>"
<%- else -%>
MEMCACHED_PARAMS="<%= result.join(' ') %>"
...
MEMCACHED_USER="<%= @user %>"

...
<%- end -%>
```

2.2.2 模板标签

首先，在ERB模板中，标签(tag)是一个重要的概念。例如：

- `<% CODE %>` 以成对出现，表示这是一段可执行代码
- `<%= EXPRESSION %>` 以成对出现，表示是插入值的表达式
- `<%# COMMENT %>` 成对出现，表示为一段注释
- `<%%` 或 `%%>` ，表示 `<%` 或 `%>` 字符

如果在标签中加入 `-` 符，则会移除缩进和换行。

在 `memcached_sysconfig.erb` 代码片段中，以下为插入值的表达式，最终会将$tcp_port,$user,$max_connections变量的值插入到Memcached的配置文件中：

```
PORT="<%= @tcp_port %>"
USER="<%= @user %>"
MAXCONN="<%= @max_connections %>"
```

而下述代码则为一段可执行代码，用于判断 `$osfamily` 的值是否为'Suse'：

```
<%- if scope['osfamily'] != 'Suse' -%>
```

2.2.3 模板变量

模板可以访问Puppet中的变量，在模板中访问变量时会有一个范围(scope)的概念，调用该模板的class或define中的变量为该模板的局部变量，可以直接使用变量名进行调用。

在ERB模板中有两种方式来访问变量：

- `@variable`
- `scope['variable']`

`@variable` 方式是ERB模板中变量的命名规范，以 `@` 开头，用于访问当前范围内的变量。如下述代码片段中，在渲染该模板文件时，Puppet会在 `class memcached` 中去搜寻与 `@tcp_port` 对应的 `$tcp_port` 变量，查询到该变量的默认值是11211，最终在 `/etc/sysconfig/memcached` 中得到配置 `PORT=11211` 。

```
PORT="<%= @tcp_port %>"
```

而 `scope['variable']` 方式则可以访问所有的变量（包括当前范围以外的），使用哈希风格的表达式用于指定需要访问的变量，例如 `scope[foo::bar]` 如下述代码片段中，Puppet将从Facter中获取到变量$osfamily的值。

```
<%- if scope['osfamily'] != 'Suse' -%>
```

本节对于模板的介绍就到这里，后续会再次谈到模板。

推荐阅读

- https://docs.puppet.com/puppet/4.10/lang_template_erb.html
- https://docs.puppet.com/puppet/4.10/lang_template.html

动手练习

1. 将 `Memcached` 服务的默认进程数为服务器核数的一半
2. 关闭 `puppet-memcached` 对防火墙规则的管理

`puppet-sysctl` 模块

`sysctl` 命令被用于在内核运行时动态地修改内核的运行参数，系统可用的内核参数可在目录/proc/sys中查询。它包含了一些TCP/IP堆栈和虚拟内存系统的高级选项， 这可以让系统管理员提高对操作系统的性能进行调优。

本节要谈的 `puppet-sysctl` 模块是由个人维护的项目，其目的是在Puppet中提供管理sysctl的接口。

`puppet-sysctl` 项目地址：https://github.com/duritong/puppet-sysctl

1.先睹为快

不想看下面大段的代码解析，已经跃跃欲试了？

OK，我们开始吧！

打开虚拟机终端并输入以下命令：

```
$ puppet apply -e 'sysctl::value { "net.ipv4.tcp_syncookies": value
```

这将开启内核中的 `tcp_syscookieds` 参数，常用于抵御 `synflood` 攻击。

我们打开 `puppet-sysctl` 模块下 `manifests/base.pp` 文件来一探究竟吧。

2.代码解析

2.1 `class sysctl::base`

`sysctl::base` 是该模块仅有的一个类，其中的代码逻辑也非常简单，仅对 `/etc/sysctl.conf` 文件的所有者和权限进行了管理。

```
class sysctl::base {
  file { '/etc/sysctl.conf':
    ensure => 'present',
    owner  => 'root',
    group  => '0',
    mode   => '0644',
  }
}
```

2.2 `define sysctl::value`

`define sysctl::value` 用于管理 `/etc/sysctl.conf` 文件中的配置项。这里，有三处值得展开解析。

```
define sysctl::value (
  $value,
  $key    = $name,
  $target = undef,
) {
  require sysctl::base
  $val1 = inline_template("<%= String(@value).split(/[\s\t]/).rejec

  sysctl { $key :
    val    => $val1,
    target => $target,
    before => Sysctl_runtime[$key],
  }
  sysctl_runtime { $key:
    val => $val1,
  }
}
```

2.2.1 `require` 函数

在 `define sysctl::value` 出现了 `require` 函数(注意：与 `require` 元参数不同)，该函数可以声明一个或多个类，并与含有此函数的容器形成依赖关系。

```
require sysctl::base
```

在该例中，Puppet在执行 `sysctl::value` 实例前，确保 `class sysctl::base` 的所有资源已被应用。

2.2.2 `inline_template` 函数

在 `define sysctl::value` 中出现了以下一段代码：

```
$val1 = inline_template("<%= String(@value).split(/[\s\t]/).reject(
```

`inline_template` 函数和 `template` 函数类似，可以简单地认为是只有一个字符串的模板。 `inline_template` 对模板求值后，生成字符串，常用于复杂的字符串拼接。

在前文中，已经提到标签 `<%= %>` 是插入值表达式，在该表达式中：首先对 `$value` 做字符串格式转换，然后以正则表达式 `\s` 和 `\t` 匹配进行字符串切割，去除空数组，对数组flatten操作，再做字符串连接。

2.2.3 自定义资源类型

Puppet中有大量内置的资源类型，如 `user` , `package` 等等，同时用户也可以通过规范进行扩展，上述代码中的 `sysctl` 和 `syscyl_runtime` 是 `puppet-sysctl` 模块中自定义的资源类型，我们会在后文中详细讲解其代码结构和实现等细节。

2.3 `class sysctl::values`

`sysctl::values` 的代码同样也非常简洁，主要是对 `sysctl::value` 进行了封装：

```
class sysctl::values($args, $defaults = {}) {
  create_resources(sysctl::value, $args, $defaults)
}
```

2.3.1 `create_resources` 函数

`create_resources` 函数接受一个hash类型的参数，将其转换为一个资源集合并添加到catalog中。这么讲比较抽象，我们可以来看一个实际的例子。

假设，接到其他部门的需求，需要在线上开启Linux内核的IP转发功能，要在开启该功能的节点上声明两个 `sysctl::value` 实例。

```
sysctl::value { 'net.ipv4.ip_forward':
  value => 1
}
sysctl::value { 'net.ipv6.conf.all.forwarding':
  value => 1
}
```

然而，在事前运维工程师是无法知道服务器需要开启哪些内核参数，按照第一章的 `理解Hiera` 一节中提到的节点数据不应该和节点数据放在一起，下面看如何借助 `create_resources` 函数来解决这个问题。 在节点定义文件中预先加入:

```
include ::sysctl::values
```

接着在 `common.yaml` Hiera文件中加入：

```
---
sysctl::values:args:
 net.ipv4.ip_forward:
   value: 1
 net.ipv6.conf.all.forwarding:
   value: 1
```

现在只需要对 `sysctl::values:args` 参数进行改动就能实现动态地管理服务器上的所有内核参数了！

3.扩展阅读

- https://docs.puppet.com/puppet/latest/function.html#createresources
- https://docs.puppet.com/puppet/4.10/lang_template.html#with-a-template-string-inlinetemplate-and-inlineepp
- https://docs.puppet.com/puppet/4.10/lang_classes.html#using-require

4.动手练习

1. 设置 `net.ipv4.tcp_timestamps` 参数为0
2. 设置 `net.ipv4.tcp_rmem` 参数为 `4096` `131072` `131072` (多个值)

`puppet-rsync` 模块

`Rsync` (remote sync)是一款通过网络进行数据同步的软件，由于Rsync会对需要同步的源和目的进行对比，只同步有改变的部分，所以相比常见的scp命令更加高效。

`puppet-rsync` 模块由puppet官方维护的项目，用于管理rsync的客户端、服务器，命令行的配置。 `puppet-rsync` 项目地址：https://github.com/puppetlabs/puppetlabs-rsync

1.先睹为快

不想看下面大段的代码解析，已经跃跃欲试了？

OK，我们开始吧！

打开虚拟机终端并输入以下命令：

```
$ puppet apply -e "class { 'rsync': }"
```

以上命令会在目标服务器上安装rsync软件包。

接下来，我们打开 `puppet-sysctl` 模块下 `manifests/base.pp` 文件来一探究竟吧。

2.代码讲解

2.1 `class rsync`

在 `class rsync` 中，除了 `package` 资源对 `rsync` 进行了声明以外，还有上一节提到的 `create_resources` 函数，分别传递了 `rsync::put` 和 `rsync::get` 参数。

```
class rsync(
  $package_ensure    = 'installed',
  $manage_package    = true,
  $puts              = {},
  $gets              = {},
) {

  if $manage_package {
    package { 'rsync':
      ensure => $package_ensure,
    } -> Rsync::Get<| |>
  }

  create_resources(rsync::put, $puts)
  create_resources(rsync::get, $gets)
}
```

rsync命令行下有两种不同的执行模式：pull和push。在 `puppet-rsync` 模块中 `define rsync::put` 对应push模式， `define rsync::get` 则对应pull模式。下面来看相关的两段代码示例。

```
# rsync push模式
rsync::put { '${rsyncDestHost}:/repo/foo':
  user    => 'user',
  source  => "/repo/foo/",
}

#rsync pull模式
rsync::get { '/foo':
  source  => "rsync://${rsyncServer}/repo/foo/",
  require => File['/foo'],
}
```

2.2 `class: rsync::server`

`class rsync::server` 则用于管理rsync server， `rsync` 在server模式下以守护进程存在，能够接收客户端的数据请求。使用时，可以在客户端使用rsync命令把文件发送到服务器端，也可以向服务器请求获取文件。 `class rsync::server` 使用xinetd来管理rsync服务，使用 `concat` 模块来管理rsync配置文件。我们会在下一节谈到 `puppet-xinetd` 模块。

```
class rsync::server(
  $use_xinetd = true,
  $address    = '0.0.0.0',
  $motd_file  = 'UNSET',
  $use_chroot = 'yes',
  $uid        = 'nobody',
  $gid        = 'nobody',
  $modules    = {},
) inherits rsync {

  $conf_file = $::osfamily ? {
    'Debian' => '/etc/rsyncd.conf',
    'suse'   => '/etc/rsyncd.conf',
    'RedHat' => '/etc/rsyncd.conf',
    default  => '/etc/rsync.conf',
  }
  ...

  if $use_xinetd {
    include xinetd
    xinetd::service { 'rsync':
      bind        => $address,
      port        => '873',
      server      => '/usr/bin/rsync',
      server_args => "--daemon --config ${conf_file}",
      require     => Package['rsync'],
    }
  ...

  concat { $conf_file: }

  concat::fragment { 'rsyncd_conf_header':
    target  => $conf_file,
    content => template('rsync/header.erb'),
    order   => '00_header',
  }
  ...
}
```

在代码片段中，出现了`inherits`关键字，与面向对象编程中的继承概念类似，其允许某个指定类从另一个类中扩展其功能和参数。

为了让读者更易于理解，称：被继承的类为`基类`，在基类上建立的新类称为`派生类`。

在使用`inherits`关键字时，将产生以下效果：

- 当派生类被声明时，基类在此之前自动被声明
- 基类成为派生类的父作用域(parent scope)，派生类将拥有基类所有的参数和资源
- 派生类可以重写基类中任何资源的属性

在此此例中，派生类`rsync::server`继承了基类`rsync`，得到了管理rsync软件包的package资源，得到了$package_ensure等参数，若要在`rsync::server`中使用该参数，则其作用域为：`$rsync::package_ensure`。需要注意的是，`inherits`的使用需要谨慎，尤其是多层继承时，会严重影响代码的可读性。在最佳实践中，仅推荐用于获取`class param`中的参数时使用，例如：

```
class example (
  String $my_param = $example::params::myparam
) inherits example::params
  { ... }
```

2.3 `define rsync::server::module`

`rsync::server::module`用于设置rsync服务实例，代码实现比较简单，以下看一段示例：`class swift::ringserver`，通过声明了`rsync::server`和`rsync::server::module`来搭建同步ring文件的rsync服务器:

```
class swift::ringserver(
  $local_net_ip,
  $max_connections = 5
) {

  include ::swift::deps
  Class['swift::ringbuilder'] -> Class['swift::ringserver']

  if !defined(Class['rsync::server']) {
    class { '::rsync::server':
      use_xinetd => true,
      address    => $local_net_ip,
      use_chroot => 'no',
    }
  }

  rsync::server::module { 'swift_server':
    path            => '/etc/swift',
    lock_file       => '/var/lock/swift_server.lock',
    uid             => 'swift',
    gid             => 'swift',
    max_connections => $max_connections,
    read_only       => true,
  }
}
```

在 `rsync::server::module {'swift_server'}` 实例中，swift_server的路径为 `/etc/swift` ，所有者和所属组是 `swift` ，设置了默认的最大连接数，设为只读权限。

2.4 `class rsync::repo`

`class rsync::repo` 对 `rsync::server::module` 进行了简单的封装，为用户创建一个存放数据的rsync仓库，包括用于存放数据的目录和rsync服务。

```
class rsync::repo {
  include rsync::server
  $base = '/data/rsync'
  file { $base:
    ensure  => directory,
  }
  # setup default rsync repository
  rsync::server::module { 'repo':
    path    => $base,
    require => File[$base],
  }
```

3.扩展阅读

- 动态域的使用 https://docs.puppet.com/puppet/5.0/lang_scope.html#dynamic-scope
- 关于继承的使用 https://docs.puppet.com/puppet/5.0/lang_classes.html#inheritance

4.动手练习

1.请说明上述代码中的 `$conf_file = $::osfamily ? { ...}` 的作用和用法

2.搭建一个用于同步软件包的rsync服务器，incoming_chmod设为'0755'，outgoing_chmod设为'0644'，只读权限。

`puppet-xinetd` 模块

1. 先睹为快
2. 代码讲解
3. 扩展阅读
4. 动手练习

xinetd是一个运行于类Unix操作系统的开放源代码的超级服务器（Super-server）守护进程。 它的功能是管理网络相关的服务。 xinetd提供类似于inetd+tcp_wrapper的功能，由于其较高的安全性，xinetd开始逐渐取代inetd。xinetd监听来自网络的请求，从而启动相应的服务。

`puppetlabs-xinetd` 模块是由puppet官方维护的项目，用于管理xinetd服务。
`puppetlabs-xinetd` 项目地址：https://github.com/puppetlabs/puppetlabs-xinetd

1.先睹为快

不想看下面大段的代码说明，已经跃跃欲试了？

Ok，我们开始吧！

打开虚拟机终端并输入以下命令：

```
$  puppet apply -e "include ::xinetd"
```

命令执行完成后，Puppet完成了对xinetd安装和配置，并启动了xinetd进程。

2.代码讲解

2.1 `class xinetd`

`class xinetd` 用于完成对xinetd的软件包的安装、配置文件的生成、服务的管理，代码比较简单，这里不再赘述。其中值得一提的是，在代码块的首段是关于文件的资源声明语句，其首字母大写的 `File` ，这与通常的 `file` 资源有何区别？

```
File {
  owner   => 'root',
  group   => '0',
  notify  => Service[$service_name],
  require => Package[$package_name],
}
```

这种以资源类型的首字母大写开头并且没有title的声明方式称为资源默认属性声明（Resource default statements），通过这种方式可以声明指定资源的默认属性。

在以上代码中， `class xinetd` 下所有的文件资源的默认属性被设置为:

- 所有者为root
- 所属组为0(即root)
- 文件发生变化将通知xinetd服务重启
- 文件被管理前，需安装xinetd软件包

所以在 `class xinted` 出现的其他 `file` 资源的相关属性将以上默认值，例如:

```
file { $conffile:
  ensure => file,
  mode => '0644',
  content => template('xinetd/xinetd.conf.erb'),
}
# 等价于：
file { $conffile:
  ensure  => file,
  mode    => '0644',
  content => template('xinetd/xinetd.conf.erb'),
  owner => 'root',
  group => '0',
  notify => Service[$service_name],
  require => Package[$package_name],
}
```

那么通过资源默认属性声明的方式，可以带来两点好处：

- 确保了相同资源默认属性的一致性
- 提高了代码复用

需要注意的是，资源默认属性声明的作用范围很大，如果你在某个类中使用了它，那么可能会对其他类或者定义产生影响，

因此，最佳实践是：只在 `site.pp` 中使用资源默认属性声明。

2.2 define xinetd::service

回顾一下，在上一节 `puppet-rsync` 模块中，类 `rsync::server` 声明了 `xinetd::service` 定义，用于创建某个rsync服务的xinetd的配置文件：

```
xinetd::service { 'rsync':
  bind        => $address,
  port        => '873',
  server      => '/usr/bin/rsync',
  server_args => "--daemon --config ${conf_file}",
  require     => Package['rsync'],
}
```

以上代码在xinetd中创建rsync服务的配置，指定了：

- 服务的监听地址
- 服务的运行端口
- 服务的运行命令
- 服务运行命令的参数
- 服务运行的依赖

扩展阅读

- 资源默认属性声明 https://docs.puppet.com/puppet/4.10/lang_defaults.html

动手练习

1. nagios是流行的开源监控项目，请使用Puppet部署nagios服务，并且通过xinted来管理nagios进程。

参考链接：https://github.com/example42/puppet-nagios

`puppet-rabbitmq` 模块

1. 先睹为快
2. 代码讲解
3. 扩展阅读
4. 动手练习

RabbitMQ是RabbitMQ Technologies Ltd开发的AMQP（Advanced Message Queue Protocol）的开源实现。RabbitMQ组件也是此书的重点章节，因为它与每个OpenStack服务息息相关，那么RabbitMQ解决了什么问题？

对于一个复杂的分布式系统而言，它包含了大量的组件或者子系统，那么这些组件之间是如何进行通信的呢？

分布式系统，顾名思义，其组件是运行在不同的服务器上，而传统的应用软件往往使用管道，信号，报文等方式来解决进程间的协作，这些进程间通讯IPC通常只是运行在单个操作系统上，不具备扩展的能力；如果使用Socket将服务组件部署到不同的服务器，需要解决以下问题：

1）消息的发送方和接收方如何维持连接，如果连接中断，如何处理这期间的已接收的数据？

2）如何解耦发送方和接收方？

3）如何有效地分发和接收消息？

4）如何实现消息处理能力的水平扩展？

5）如何保证接收方接收到了完整，正确的数据？

高级消息队列协议(AMQP)解决了上述问题，而RabbitMQ用Erlang实现了一个异步，模块化，可扩展的高级消息队列协议。

`puppet-rabbitmq` 是由Puppet官方维护的模块，用于管理RabbitMQ服务的安装，配置。

`puppet-rabbitmq` 项目地址：https://github.com/voxpupuli/puppet-rabbitmq

1.先睹为快

不想看下面大段的代码说明，已经跃跃欲试了？

OK，我们开始吧！

打开虚拟机终端并输入以下命令：

```
$ puppet apply -e "class { 'rabbitmq': }"
```

等待上述命令执行完成，Puppet完成了对RabbitMQ服务的安装和配置。

2.代码讲解

2.1 `class rabbitmq`

`class rabbitmq` 是一个入口类，用于声明当前模块中的相关资源，同时也会包含一些逻辑判断和声明，如：判断参数值类型是否符合预期、调用其它类（include）、继承params类、判断参数是否启用LADP验证等等。

```
class rabbitmq(
  $admin_enable                = $rabbitmq::params::admin_enable,
  $cluster_nodes               = $rabbitmq::params::cluster_nodes,
  $config                      = $rabbitmq::params::config,
  $config_cluster              = $rabbitmq::params::config_cluster,
  ...
)inherits rabbitmq::params {
  validate_re($package_apt_pin, '^(|\d+)$')
  ...
  include '::rabbitmq::install'
  include '::rabbitmq::config'
  include '::rabbitmq::service'
  include '::rabbitmq::management'

  if $admin_enable and $service_manage {
    include '::rabbitmq::install::rabbitmqadmin'

    rabbitmq_plugin { 'rabbitmq_management':
      ensure   => present,
      require  => Class['rabbitmq::install'],
      notify   => Class['rabbitmq::service'],
      provider => 'rabbitmqplugins',
    }

    Class['::rabbitmq::service'] -> Class['::rabbitmq::install::rak
    Class['::rabbitmq::install::rabbitmqadmin'] -> Rabbitmq_exchang
  }
  ...

}
```

在代码块首有一些类似于`validate_re($package_apt_pin, '^(|\d+)$')`的代码，其实`validate_re`函数接收两个参数：参数名，正则表达式。用于检查指定参数的传入值是否与给定的正则表达式匹配。 因此，在应用catalog前对输入数据进行检查，可以提前发现的用户错误传参。

`class rabbitmq`作为一个入口类，声明了4个类：

- rabbitmq::install

- rabbitmq::config
- rabbitmq::service
- rabbitmq::management

2.2 class rabbitmq::install

`rabbitmq::install` 用于管理RabbitMQ Server的软件部署和配置，注意其参数的默认值是 `$rabbitmq::param_name` 的格式，说明该类在被声明时， `class rabbitmq` 也需同时被声明。

```
class rabbitmq::install {
  $package_ensure   = $rabbitmq::package_ensure
  $package_name     = $rabbitmq::package_name
  $package_provider = $rabbitmq::package_provider
  $package_require  = $rabbitmq::package_require
  $package_source   = $rabbitmq::real_package_source

  package { 'rabbitmq-server':
    ensure   => $package_ensure,
    name     => $package_name,
    provider => $package_provider,
    notify   => Class['rabbitmq::service'],
    require  => $package_require,
  }

  if $package_source {
    Package['rabbitmq-server'] {
      source  => $package_source,
    }
  }

  if $rabbitmq::environment_variables['MNESIA_BASE'] {
    file { $rabbitmq::environment_variables['MNESIA_BASE']:
      ensure  => 'directory',
      owner   => 'root',
      group   => 'rabbitmq',
      mode    => '0775',
      require => Package['rabbitmq-server'],
    }
  }
}
```

2.3 `class rabbitmq::config`

`rabbitmq::config` 类用于统一管理RabbitMQ服务的目录和配置文件。可能会有读者有疑问，为什么要将软件包的安装和配置文件的管理分拆为 两个类。原因很简单，为了代码可读性， `rabbitmq::config` 的代码长度有两百多行，若与其他代

码合并在一起，阅读起来会非常痛苦。这也是Puppet的最佳实践之一：尽可能保持代码的简洁和可读性。

```
class rabbitmq::config {

  $admin_enable                 = $rabbitmq::admin_enable
  $cluster_node_type            = $rabbitmq::cluster_node_type
  $cluster_nodes                = $rabbitmq::cluster_nodes
  $config                       = $rabbitmq::config
  ...
  }
  ...
  file { '/etc/rabbitmq':
    ensure => directory,
    owner  => '0',
    group  => '0',
    mode   => '0644',
  }
  file { '/etc/rabbitmq/ssl':
    ensure => directory,
    owner  => '0',
    group  => '0',
    mode   => '0644',
  }
  ...
```

2.4 `class rabbitmq::service`

`rabbitmq::install` 和 `rabbitmq::config` 分别完成了软件包的安装、配置文件的生成，准备工作已经完成，`rabbitmq::service` 类用于管理服务状态。

```
class rabbitmq::service(
  Enum['running', 'stopped'] $service_ensure  = $rabbitmq::service_
  Boolean $service_manage                     = $rabbitmq::service_
  $service_name                               = $rabbitmq::service_
) inherits rabbitmq {

  if ($service_manage) {
    if $service_ensure == 'running' {
      $ensure_real = 'running'
      $enable_real = true
    } else {
      $ensure_real = 'stopped'
      $enable_real = false
    }

    service { 'rabbitmq-server':
      ensure     => $ensure_real,
      enable     => $enable_real,
      hasstatus  => true,
      hasrestart => true,
      name       => $service_name,
    }
  }
}
```

读者可能已经注意到参数$service_ensure和$service_manage被声明了数据类型，其中Boolean被称为是数据类型(Data types)，在Puppet中有以下数据类型:

- Strings
- Numbers
- Booleans
- Arrays
- Hashes
- Regular Expressions
- Sensitive
- Undef
- Resource References

- Default

此外，Enum称为是抽象数据类型(abstract data types)，可以灵活地匹配/限制指定参数的数据类型。

例如，`Boolean $service_manage` 严格地限定了$service_manage的数据类型为布尔型，而使用 `Optional[String, Boolean] $service_manage` 则可以指定$service_manage的数据类型可以是布尔型或者字符串。

3.扩展阅读

- 数据类型 https://docs.puppet.com/puppet/4.10/lang_data.html
- 抽象数据类型 https://docs.puppet.com/puppet/4.10/lang_data_abstract.html

4.动手练习

1.默认安装的时候有guest用户，出于安全考虑，会删除此用户，请使用puppet-rabbitmq完成此操作。 2.如何使用自定义资源rabbitmq_user来创建用户？ 3.在OpenStack中，fanout类型的队列应在程序退出时删除，RabbitMQ中可以使用Policy设置Queue的TTL，请使用rabbitmq_policy将.*上所有queue的ttl设置为18000s。

`puppet-firewall` 模块

iptables是一个配置Linux内核防火墙的命令行工具，通过设定一些特殊的规则，以允许或拒绝数据包通过。

`puppet-firewall` 模块是由Puppet公司维护的官方模块，用于管理防火墙和其规则。该模块通过扩展自定义资源类型来管理firewall规则和 iptables chains，当前支持iptables和ip6tables。

`puppet-firewall` 项目地址：'https://github.com/puppetlabs/puppetlabs-firewall'

1.先睹为快

不想看下面大段的代码解析，已经跃跃欲试了？

先别激动，这个模块的使用是有风险的，操作不慎会把自己也拒之门外，请确保可以通过除ssh外的方式登陆。

创建learn_firewall.pp文件并编辑:

```
class my_fw::pre {
  Firewall {
    require => undef,
  }

  # Default firewall rules
  firewall { '000 accept all icmp':
    proto  => 'icmp',
    action => 'accept',
  }->
  firewall { '001 accept all to lo interface':
    proto   => 'all',
    iniface => 'lo',
```

```
    action  => 'accept',
  }->
  firewall { '002 reject local traffic not on loopback interface
    iniface     => '! lo',
    proto       => 'all',
    destination => '127.0.0.1/8',
    action      => 'reject',
  }->
  firewall { '003 accept related established rules':
    proto  => 'all',
    state  => ['RELATED', 'ESTABLISHED'],
    action => 'accept',
  }
}

class my_fw::post {
    firewall { '999 drop all':
      proto  => 'all',
      action => 'drop',
      before => undef,
    }
}

class my_fw {
  firewall { '004 Allow inbound SSH':
    dport    => 22,
    proto    => tcp,
    action   => accept,
    provider => 'iptables',
  }
  firewall { '005 Allow inbound HTTP':
    dport    => 80,
    proto    => tcp,
    action   => accept,
    provider => 'iptables',
  }
}

Firewall {
  before  => Class['my_fw::post'],
```

```
    require => Class['my_fw::pre'],
  }

  class { ['my_fw::pre', 'my_fw::post','my_fw']: }

  class { 'firewall': }
```

在终端下输入以下命令：

```
$ puppet apply -v learn_firewall.pp
```

在执行该命令前,操作系统的防火墙规则为空：

```
$ iptables -L

Chain INPUT (policy ACCEPT)
target     prot opt source               destination

Chain FORWARD (policy ACCEPT)
target     prot opt source               destination

Chain OUTPUT (policy ACCEPT)
target     prot opt source               destination
```

在执行完成该命令后，防火墙规则发生了以下变化：

```
$ iptables -L

Chain INPUT (policy ACCEPT)
target     prot opt source               destination
ACCEPT     icmp --  anywhere             anywhere             /* 00
ACCEPT     all  --  anywhere             anywhere             /* 00
REJECT     all  --  anywhere             loopback/8           /* 00
ACCEPT     all  --  anywhere             anywhere             state
ACCEPT     tcp  --  anywhere             anywhere             multi
ACCEPT     tcp  --  anywhere             anywhere             multi
ACCEPT     all  --  anywhere             anywhere             state
ACCEPT     icmp --  anywhere             anywhere
ACCEPT     all  --  anywhere             anywhere
ACCEPT     tcp  --  anywhere             anywhere             state
REJECT     all  --  anywhere             anywhere             rejec
DROP       all  --  anywhere             anywhere             /* 99

Chain FORWARD (policy ACCEPT)
target     prot opt source               destination
REJECT     all  --  anywhere             anywhere             rejec

Chain OUTPUT (policy ACCEPT)
target     prot opt source               destination
```

2.代码讲解

2.1 `class firewall`

`firewall` 类用于管理Iptables软件包和服务，会根据内核类型申明不同的类进行管理。

```
class firewall (
  $ensure       = running,
  $pkg_ensure   = present,
  $service_name = $::firewall::params::service_name,
  $package_name = $::firewall::params::package_name,
) inherits ::firewall::params {
  case $ensure {
    /^(running|stopped)$/: {
      # Do nothing.
    }
    default: {
      fail("${title}: Ensure value '${ensure}' is not supported")
    }
  }

  case $::kernel {
    'Linux': {
      class { "${title}::linux":
        ensure       => $ensure,
        pkg_ensure   => $pkg_ensure,
        service_name => $service_name,
        package_name => $package_name,
      }
    }
    'FreeBSD': {
    }
    default: {
      fail("${title}: Kernel '${::kernel}' is not currently support
    }
  }
}
```

以Linux为例，`firewall::linux`会根据操作系统的不同调用对应的firewall::linux::xxx类:

```
case $::operatingsystem {
  'RedHat', 'CentOS', 'Fedora', 'Scientific', 'SL', 'SLC', 'Ascer
  'CloudLinux', 'PSBM', 'OracleLinux', 'OVS', 'OEL', 'Amazon', 'X
    class { "${title}::redhat":
      ensure       => $ensure,
      enable       => $enable,
      package_name => $package_name,
      service_name => $service_name,
      require      => Package['iptables'],
    }
  }
  'Debian', 'Ubuntu': {
    class { "${title}::debian":
      ensure       => $ensure,
      enable       => $enable,
      package_name => $package_name,
      service_name => $service_name,
      require      => Package['iptables'],
    }
  }
  ...
  }
```

以RedHat为例，`firewall::linux::redhat`会根据操作系统和版本的不同跳转到相应的逻辑。通过这个模块可以发现，`firewall`类仅完成了安装软件包和管理服务状态，但要维护一个支持多平台和版本的模块并非易事，需要投入大量的精力进去，这也是社区模式可以得到众多公司认可的原因。

```
# RHEL 7 and later and Fedora 15 and later require the iptables-s
# package, which provides the /usr/libexec/iptables/iptables.init
# lib/puppet/util/firewall.rb.
if ($::operatingsystem != 'Amazon')
and (($::operatingsystem != 'Fedora' and versioncmp($::operatings
or  ($::operatingsystem == 'Fedora' and versioncmp($::operatingsy
  service { 'firewalld':
    ensure => stopped,
    enable => false,
```

```
        before => Package[$package_name],
      }
    }

    if $package_name {
      package { $package_name:
        ensure => $package_ensure,
        before => Service[$service_name],
      }
    }

    if ($::operatingsystem != 'Amazon')
    and (($::operatingsystem != 'Fedora' and versioncmp($::operatings
    or  ($::operatingsystem == 'Fedora' and versioncmp($::operatingsy
    ...
    # Redhat 7 selinux user context for /etc/sysconfig/iptables is se
    case $::selinux {
      #lint:ignore:quoted_booleans
      'true',true: {
        case $::operatingsystemrelease {
          /^(6|7)\..*/: { $seluser = 'unconfined_u' }
          default: { $seluser = 'system_u' }
        }
      }
      #lint:endignore
      default:     { $seluser = undef }
    }

    file { "/etc/sysconfig/${service_name}":
      ensure  => present,
      owner   => 'root',
      group   => 'root',
      mode    => '0600',
      seluser => $seluser,
    }
  }
```

在上述代码中，需要理解以下新知识点：

第一点， `versioncmp` 函数用于比较两个版本号并返回比较结果，例如:

```
$result = versioncmp(a, b)
```

- a大于b，返回1
- a等于b，返回0
- a小于b，返回-1

第二点，要理解运算的优先级顺序，在上述代码出现了一段比较复杂的条件语句:

```
if ($::operatingsystem != 'Amazon')
  and (($::operatingsystem != 'Fedora' and versioncmp($::operatings
  or  ($::operatingsystem == 'Fedora' and versioncmp($::operatingsy
```

首先()的优先级最高，因此以下表达式会优先进行计算：

- ($::operatingsystem != 'Amazon')
- (($::operatingsystem != 'Fedora' and versioncmp($::operatingsystemrelease, '7.0') >= 0))
- ($::operatingsystem == 'Fedora' and versioncmp($::operatingsystemrelease, '15') >= 0))

其次， `==` 的优先级等于 `!=` 高于 `>=` 高于 `and` 。最后是最外层的and/or运算 `if statement1 and statement2 or statement 3` ，那么其运算顺序是哪一种?

- if (statement1 and statement2) or (statement 3)
- if statement1 and (statement2 or statement 3)

答案是前一种，因为 `and` 的优先级高于 `or`

第三点，掌握case条件语句的语法。

case条件语句和if条件语句类似，均是选择其中的一个Puppet代码块进行执行，但其更适合用于字符串和数值的匹配。

```
case $facts['name'] {
  'A':        { include role::case1 }
  'B', 'C':  { include role::case2  }
  /^(D|E)$/: { include role::case3  }
  default:   { include role::default_case }
}
```

2.2 `type firewall`

资源类型 `firewall` 用于管理防火墙规则，以下举例说明如何在真实环境中使用该类型：

2.2.1 为apache开启80和443端口

```
firewall { '100 allow http and https access':
    dport  => [80, 443],
    proto  => tcp,
    action => accept,
  }
```

2.2.2 丢弃FIN/RST/ACK包如果没有对应的SYN包

```
firewall { '002 drop NEW external website packets with FIN/RST/ACK
  chain     => 'INPUT',
  state     => 'NEW',
  action    => 'drop',
  proto     => 'tcp',
  sport     => ['! http', '! 443'],
  source    => '! 10.0.0.0/8',
  tcp_flags => '! FIN,SYN,RST,ACK SYN',
}
```

2.2.3 SNAT 10.1.2.0/24

```
firewall { '100 snat for network foo2':
  chain    => 'POSTROUTING',
  jump     => 'MASQUERADE',
  proto    => 'all',
  outiface => 'eth0',
  source   => '10.1.2.0/24',
  table    => 'nat',
}
```

2.3 `type firewallchain`

资源类型 `firewallchain` 用于管理管理防火墙的规则链，以下举例说明如何在真实环境中使用该类型:

2.3.1 默认丢弃INPUT链上的包

```
  firewallchain { 'INPUT:filter:IPv4':
    ensure => present,
    policy => drop,
    before => undef,
  }
```

需要说明的是，这个模块有一定的 局限性，如只支持管理iptable和ip6tables。此外，在和Neutron同时使用时会遇到iptable规则的冲突问题。

3.扩展阅读

- 运算符优先级
  https://docs.puppet.com/puppet/4.10/lang_expressions.html#order-of-operations
- case条件语句
  https://docs.puppet.com/puppet/4.10/lang_conditional.html#case-statements

4.动手练习

1. 本章给出的第一个示例learn_firewall.pp存在一些问题,当修改防火墙规则时，旧的规则不会被删除，请修复这个问题
2. 为OpenStack Nova服务编写firewall规则，开放相应的服务端口

puppet-mysql

几乎所有OpenStack核心组件都会用到数据库组件, OpenStack支持的数据库后端有SQlite, MySQL, PostGreSQL。

而MySQL是使用最广泛的关系型数据库管理系统(Relational Database Management System：关系数据库管理系统), 数据库服务是OpenStack的基础服务，在部署和维护OpenStack集群之前，须要了解数据库相关的知识。`puppet-mysql` 模块是由Puppet官方所维护的项目，用于管理MySQL客户端程序和服务端的配置，以及管理备份脚本的支持，包括用于管理MySQL数据库，用户，授权等的自定义资源。

`puppet-mysql` 项目地址：https://github.com/puppetlabs/puppetlabs-mysql

1.先睹为快

不想看下面大段的代码解析，已经跃跃欲试了？

OK，我们开始吧！

打开虚拟机终端并输入以下命令：

```
$ puppet apply -e "class { '::mysql::server': }"
```

等待终端完成命令执行后，在终端输入 `mysql` 就能直接连入MySQL服务了。

2.代码讲解

与其他模块不同的是，在puppet-mysql中并没有init.pp这个类(即 `class mysql`)。熟悉Python的读者知道在每个Python模块中含有 `__init__.py` 文件，用于将当前目录注册为Python模块。然而对于Puppet模块来说，init.pp并不是强制的，即使不

存在也不会影响Puppet识别其为Puppet模块。

2.1 `class mysql::server`

`mysql::server` 类用于MySQL服务器端的部署，配置和服务的管理，以及root用户的管理。这些功能则 是通过声明其他类来完成的。值得注意的是被申明的类的命名域是 `mysql::server::` ，将与MySQL服务器端相关的类统一放到了同个目录下(manifests/server/)。

```
include '::mysql::server::config'
include '::mysql::server::install'
include '::mysql::server::installdb'
include '::mysql::server::service'
include '::mysql::server::root_password'
include '::mysql::server::providers'
```

以上类主要完成以下操作:

- 相关配置文件的安装
- 软件包的安装
- 初始化数据库
- MySQL服务的启动
- root用户/密码的设定

2.2 `class mysql::server::installdb`

`class mysql::server::installdb` 用于MySQL数据库的初始化工作，它使用了自定义资源类型 `mysql_datadir` 来完成数据库目录的初始化：

```
mysql_datadir { $datadir:
  ensure              => 'present',
  datadir             => $datadir,
  basedir             => $basedir,
  user                => $mysqluser,
  log_error           => $log_error,
  defaults_extra_file => $_config_file,
}
```

`mysql_datadir` 资源类型实际调用了 `mysql_install_db` 命令用于完成数据库目录的初始化，资源的属性将被传入作为该命令的参数。

2.3 `class mysql::server::config`

`mysql::server::config` 类用于mysql配置文件和目录的管理，最核心的是对my.cnf文件的管理，以下代码中：

```
  if $mysql::server::manage_config_file  {
    file { 'mysql-config-file':
      path                    => $mysql::server::config_file,
      content                 => template('mysql/my.cnf.erb'),
      mode                    => '0644',
      selinux_ignore_defaults => true,
    }
```

使用erb模板的方式完成了对my.cnf文件的管理，关于erb模板在前面的章节已经说明，这里不再赘述。

2.3.1 如何动态地管理配置项?

使用模板带来的一个缺点是不灵活：所有的配置项需要提前写入到模板文件中，而模板不一定能做到包含所有的配置项和配置段落。

那么如何在生成配置文件时动态地添加配置项呢？

有多种手段来实现这个需求，我们可以先了解 `puppet-mysql` 是如何解决这个问题的。

在 `mysql::server` 类中有一个特殊的参数：

```
...
  $override_options        = {},
...
  # Create a merged together set of options.  Rightmost hashes win
  $options = mysql_deepmerge($mysql::params::default_options, $over
```

参数 `$override_options` 是一个为空的哈希字典，通过变量名称可以判断，这是参数用于重写MySQL的默认选项。 参数 `$mysql::params::default_options` 是一个含有MySQL配置项默认值的哈希字典，其默认值如下：

```
$default_options = {
  'client'          => {
    'port'          => '3306',
    'socket'        => $mysql::params::socket,
  },
  'mysqld_safe'        => {
    'nice'             => '0',
    'log-error'        => $mysql::params::log_error,
    'socket'           => $mysql::params::socket,
  },
...
```

`mysql_deepmerge` 是由 `puppet-mysql` 模块实现的自定义函数，用于对2个哈希字典执行合并操作。

例如：

```
$hash1 = {'one' => 1, 'two' => 2, 'three' => { 'four' => 4 } }
$hash2 = {'two' => 'dos', 'three' => { 'five' => 5 } }
$merged_hash = mysql_deepmerge($hash1, $hash2)
```

最终得到的结果是：$merged_hash = { 'one' => 1, 'two' => 'dos', 'three' => { 'four' => 4, 'five' => 5 } }

接下来，设置mysql::server::override_options 就可以实现动态管理配置文件的目的，例如:

```
$override_options = {
  'newsection' => {
    'item' => 'value',
  }
}
```

这个变量最终生成的my.cnf 配置文件内容将新增以下配置：

```
[newsection]
item = value
```

`define mysql::db`

`define mysql::db` 用于创建数据库，以及相关用户和密码以及权限，以下是一段代码示例：

```
mysql::db { 'mydb':
  user     => 'myuser',
  password => 'mypass',
  host     => 'localhost',
  grant    => ['SELECT', 'UPDATE'],
}
```

OpenStack服务在对其进行了封装后使用，本书会在后续的 `puppet-openstacklib` 章节中提及。

3.扩展阅读

- 编写自定义函数 https://docs.puppet.com/puppet/4.10/lang_write_functions_in_puppet.html

4.动手练习

1. 阅读mysql::server::backup代码并使用其来实现数据库备份脚本的管理
2. 请使用 `puppet-mysql` 模块创建数据库keystone

`puppet-vcsrepo` 模块

`puppet-vcsrepo` 是由Puppet公司维护的官方模块,提供了管理版本控制系统(VCS)的能力，如:git,svn,cvs,bazaar等。 `puppet-vcsrepo` 项目地址：https://github.com/puppetlabs/puppetlabs-vcsrepo

注**1** `vcsrepo` 并不会主动安装任何的vcs软件，因此在使用该模块前需要完成VCS的安装。
注**2** `git` 是Puppet公司唯一官方支持的vcs provider

1.先睹为快

不想看下面大段的代码解析，已经跃跃欲试了？

OK，我们开始吧！

创建一个git.pp文件并输入：

```
vcsrepo { '/tmp/git_repo':
  ensure   => present,
  provider => git,
}
```

打开虚拟机终端并输入以下命令：

```
$ puppet apply -v git.pp
```

该命令将会创建一个git仓库，其路径是'/tmp/git_repo'。

2.使用示例

`puppet-vcsrepo` 模块除了自定义资源类型vcsrepo以外，并没有任何manfests代码。因此，本节主要介绍使用vcsrepo来管理git仓库。

例1: 创建和管理一个空的git bare仓库：

```
vcsrepo { '/path/to/repo':
  ensure   => bare,
  provider => git,
}
```

例2：clone/pull一个repo：

```
vcsrepo { '/path/to/repo':
  ensure   => present,
  provider => git,
  source   => 'git://example.com/repo.git',
}
```

例3：指定branch或tag：

注3：默认 `vcsrepo` 会使用源仓库master分支的HEAD。若要使用其他分支或指定的commit，可以设置 `revision` 来指定branch名称或commit SHA值或者tag号

- 指定Branch:

  ```
  vcsrepo { '/path/to/repo':
  ensure   => present,
  provider => git,
  source   => 'git://example.com/repo.git',
  revision => 'development',
  }
  ```

- 指定SHA：

```
vcsrepo { '/path/to/repo':
ensure   => present,
provider => git,
source   => 'git://example.com/repo.git',
revision => '0c466b8a5a45f6cd7de82c08df2fb4ce1e920a31',
}
```

- 指定tag：

```
vcsrepo { '/path/to/repo':
ensure   => present,
provider => git,
source   => 'git://example.com/repo.git',
revision => '1.1.2rc1',
}
```

例4：保持repo为最新代码：

```
vcsrepo { '/path/to/repo':
  ensure   => latest,
  provider => git,
  source   => 'git://example.com/repo.git',
  revision => 'master',
}
```

例5：clone repo，但是跳过初始化submodule:

```
vcsrepo { '/path/to/repo':
  ensure     => latest,
  provider   => git,
  source     => 'git://example.com/repo.git',
  submodules => false,
}
```

例6：设置多个source，必须指定明确的remote：

```
vcsrepo { '/path/to/repo':
  ensure   => present,
  provider => git,
  remote   => 'origin'
  source   => {
    'origin'       => 'https://github.com/puppetlabs/puppetlabs-vcs
    'other_remote' => 'https://github.com/other_user/puppetlabs-vcs
  },
}
```

例7：使用指定用户的SSH密钥来clone repo：

若要使用SSH方式连接到源码仓库，推荐使用Puppet来管理SSH密钥，并使用 `require` 元参数来确保它们间的执行顺序。

```
csrepo { '/path/to/repo':
  ensure     => latest,
  provider   => git,
  source     => 'git://username@example.com/repo.git',
  user       => 'toto', #uses toto's $HOME/.ssh setup
  require    => File['/home/toto/.ssh/id_rsa'],
}
```

2.1 Git支持的特性和参数

特性:

- `bare_repositories`
- `depth`
- `multiple_remotes`
- `reference_tracking`
- `ssh_identity`
- `submodules`
- `user`

参数:

- `depth`
- `ensure`
- `excludes`
- `force`
- `group`
- `identity`
- `owner`
- `path`
- `provider`
- `remote`
- `revision`
- `source`
- `user`

3.动手练习

1.使用 `vcsrepo` 管理nova源码仓库，并使用stable/ocata分支
2.使用 `vcsrepo` 管理一个带有submodule的项目，并指定管理submodule

`puppet-mongodb` 模块

MongoDB是一个基于分布式文件存储的数据库，旨在为Web应用提供可扩展的高性能数据存储解决方案。 `puppetlabs-mongodb` 模块是由Puppet公司维护的官方项目，用于管理MongoDB服务，包括:

- 配置mongodb server(包括不同模式)
- 配置mongodb client
- 配置mongos
- 管理安装源

`puppet-mongodb` 项目地址：https://github.com/puppetlabs/puppetlabs-mongodb

1.先睹为快

不想看下面大段的代码解析，已经跃跃欲试了？

OK，我们开始吧！

打开虚拟机终端并输入以下命令：

```
$ puppet apply -e "include mongodb::server,mongodb::client" -v
```

在看到赏心悦目的绿字后，Puppet已经完成了MongoDB服务的安装，配置和启动，输入mongo就可以进入命令行界面了。

2.代码讲解

2.1 `class mongodb`

`class mongodb` 的代码比较简单，声明了 `class mongodb::server` 。

2.2 `class mongodb::server`

2.2.1 类包含和链式箭头

在该类中，有一段复杂的代码:

```
  if ($ensure == 'present' or $ensure == true) {
    if $restart {
      anchor { 'mongodb::server::start': }
      -> class { 'mongodb::server::install': }
      # If $restart is true, notify the service on config changes (
      -> class { 'mongodb::server::config': }
      ~> class { 'mongodb::server::service': }
      -> anchor { 'mongodb::server::end': }
    } else {
      anchor { 'mongodb::server::start': }
      -> class { 'mongodb::server::install': }
      # If $restart is false, config changes won't restart the serv
      -> class { 'mongodb::server::config': }
      -> class { 'mongodb::server::service': }
      -> anchor { 'mongodb::server::end': }
    }
  } else {
    anchor { 'mongodb::server::start': }
    -> class { '::mongodb::server::service': }
    -> class { '::mongodb::server::config': }
    -> class { '::mongodb::server::install': }
    -> anchor { 'mongodb::server::end': }
  }
```

在Puppet中,非常特别的一点是：资源和类的执行顺序并不是由上到下的。其背后是由于Puppet本身的设计机制产生了这一结果，在此我们不做展开。

而链式箭头(chain arrow)用于指定资源的执行顺序，一共有两种类型的运算符：

- 序列箭头 `->` ，左侧资源的执行顺序优于右侧

- 通告箭头 `~>` ，左侧资源执行后，右侧资源将会刷新

在 `mongodb::server` 中，若$restart为true，则执行以下代码段:

```
anchor { 'mongodb::server::start': }
-> class { 'mongodb::server::install': }
-> class { 'mongodb::server::config': }
~> class { 'mongodb::server::service': }
-> anchor { 'mongodb::server::end': }
```

表示MongoDB配置文件的变化(`mongodb::server::config`)将触发MongoDB Server端服务的重启(`mongodb::server::service`)。

若$restart为false，则执行以下代码段:

```
  anchor { 'mongodb::server::start': }
  -> class { 'mongodb::server::install': }
  -> class { 'mongodb::server::config': }
  -> class { 'mongodb::server::service': }
  -> anchor { 'mongodb::server::end': }
}
```

表示MongoDB配置文件的变更在管理MongoDB Server端服务之前。

其次，在Puppet中并无法通过include加上链式箭头声明的方式来指定类的执行顺序，或者说在类中无法通过include的方式包含(contain)一个类。

由于include和contain翻译成中文都可以理解为包含或者含有，从字面理解来看比较晦涩，我们通过举例说明。

```
class first {
  notify { 'foo': }
}

class second {
  notify { 'bar': }
}

class classa {
  include first
}

class classb {
  include second
}

Class['classa'] -> Class['classb']

include classa
include classb
```

无论将两个类之间的执行顺序如何改变，其输出结果可能是"foo bar"也可能是"bar foo"。

而anchor是解决这个问题的方法之一，其格式通常如下:

```
anchor{'start':} -> class{'new_class':} -> anchor{'end':}
```

通过这种方式使得new_class类被包含，从而可以指定类的依赖顺序。例如：

```
  anchor { 'mongodb::server::start': }
  -> class { '::mongodb::server::service': }
  -> class { '::mongodb::server::config': }
  -> class { '::mongodb::server::install': }
  -> anchor { 'mongodb::server::end': }
  }
```

其执行顺序是:

1. mongodb::server::service
2. mongodb::server::config
3. mongodb::server::install

在Puppet 3.4.0之前，使用`anchor`资源类型是解决类包含类的唯一方法。

在Puppet 3.4.0之后，新增了函数contain的方法来解决这个问题。但在使用contain声明多个class时，无法和anchor一样同时配合链式箭头使用，而需要单独声明。如:

```
class a {
  notify { 'a':}
}
class b {
  notify { 'b':}
}
class include_class {
  contain a
  contain b
  Class['a']->Class['b']
}
```

2.2.2

MongoDB分为三种模式：StandAlone，Replication和Sharding。

StandAlone是标准单机环境，Replication是主从结构，一个Primary，多个Secondary，Sharding，share nothing的结构，每台机器只存一部分数据。mongod服务器存数据，mongos服务器负责路由读写请求，元数据存在config数据库中。

创建MongoDB server时可以设置为config server或者shard server，对应的参数为configsvr或shardsvr，但是只能选择其一。

同时，在mongodb::server中可以通过replset参数来配置副本集的名称，通过replset_config或replset_members指定副本集中的成员，当然replset_members也是要转换为replset_config的。

```
$replset_config_REAL = {
 "${replset}" => {
   'ensure'   => 'present',
   'members'  => $replset_members
 }
}
```

2.3 `class mongodb::client`

`mongodb::client` 用于安装MongoDB客户端，声明了mongodb::client::install，其代码结构和mongodb::server相似。

2.4 `class mongodb::db`

`class mongodb::db` 用于创建MongoDB数据库，创建数据库时可以传入密码或者是一个hash的密码，调用方式如下：

```
mongodb::db { 'testdb':
  user          => 'user1',
  # password_hash是'user1:mongo:pass1'的md5值
  password_hash => 'a15fbfca5e3a758be80ceaf42458bcd8',
}
```

2.5 `class mongodb::mongos`

mongodb::mongos用于配置Mongo Shard进程，其代码结构和mongodb::server相似，通过声明install、config、service三个class来配置mongos,在这就不再赘述。

2.6 `class mongodb::repo`

`class mongodb::repo` 用于配置安装源，也支持通过repo_location参数自己配置安装源。

3.扩展阅读

- Containment https://docs.puppet.com/puppet/4.10/lang_containment.html
- What is Class containment https://puppet.com/blog/class-containment-puppet
- Relationships and orderings https://docs.puppet.com/puppet/5.0/lang_relationships.html#syntax-chaining-arrows

4.动手练习

1. 配置一个mongo集群，使用Replication模式
2. 配置一个mongo集群，使用sharding模式

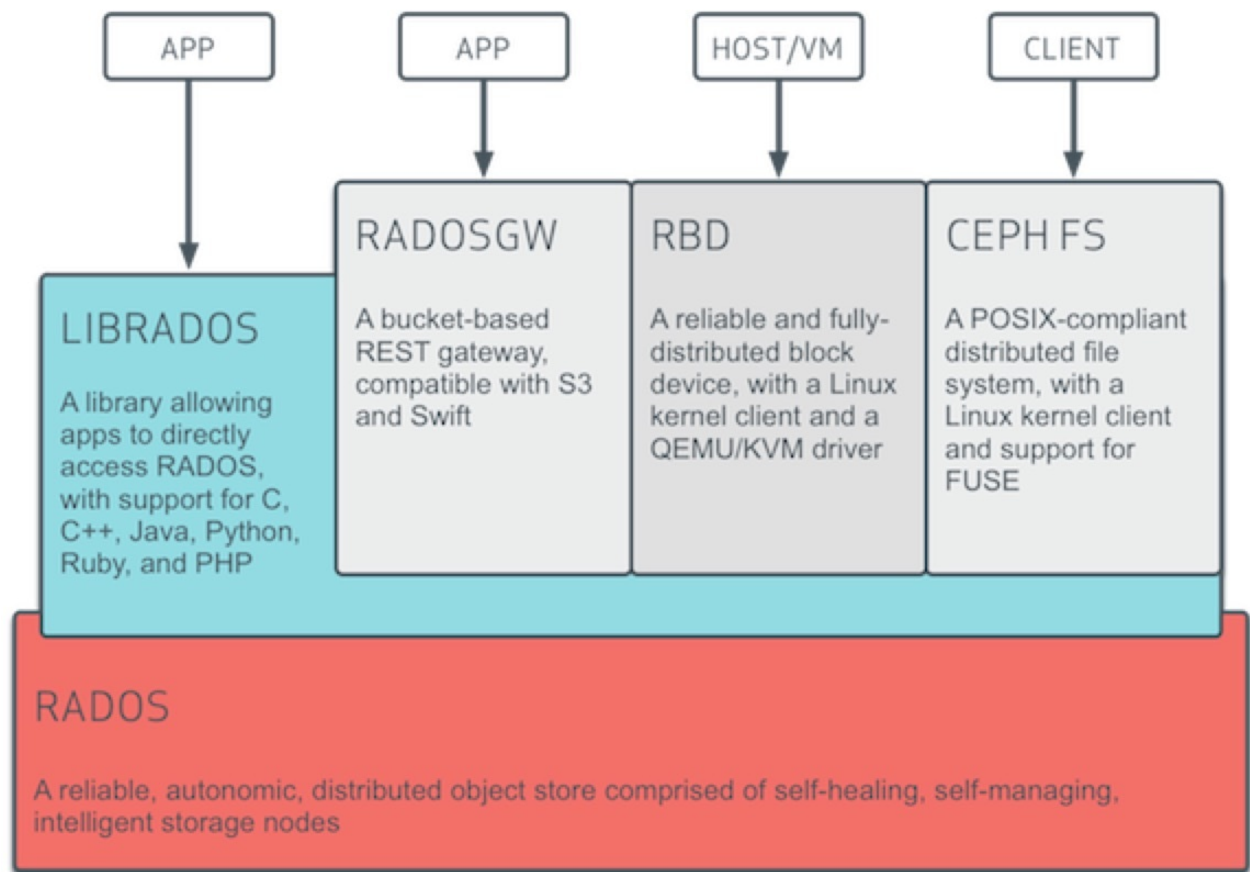
puppet-ceph

本节作者：薛飞扬

建议阅读时间 **2**小时

Ceph是一个分布式存储系统，诞生于2004年，是最早致力于开发下一代高性能分布式文件系统的项目。随着云计算的发展，ceph乘上了OpenStack的春风，进而成为了开源社区受关注较高的项目之一。

Ceph基本结构



自下向上，可以将Ceph系统分为四个层次:

1.基础存储系统RADOS（Reliable, Autonomic, Distributed Object Store，即可靠的、自动化的、分布式的对象存储） RADOS本身也是分布式存储系统，CEPH所有的存储功能都是基于RADOS实现,RADOS由大量的存储设备节点组成，每个节点拥有自己的硬件资源（CPU、内存、硬盘、网络），并运行着操作系统和文件系统。

2.基础库librados 这一层的功能是对RADOS进行抽象和封装，并向上层提供API，以便直接基于RADOS（而不是整个Ceph）进行应用开发。

3.高层应用接口 这一层包括了三个部分：RADOS GW（RADOS Gateway）、RBD（Reliable Block Device）和Ceph FS（Ceph File System），其作用是在librados库的基础上提供抽象层次更高、更便于应用或客户端使用的上层接口。 其中，RADOS GW是一个提供与Amazon S3和Swift兼容的RESTful API的gateway，以供相应的对象存储应用开发使用。RADOS GW提供的API抽象层次更高，但功能则不如librados强大。因此，开发者应针对自己的需求选择使用。 RBD则提供了一个标准的块设备接口，常用于在虚拟化的场景下为虚拟机创建volume。目前，Red

Hat已经将RBD驱动集成在KVM/QEMU中，以提高虚拟机访问性能。 Ceph FS是一个POSIX兼容的分布式文件系统。由于还处在开发状态，因而Ceph官网并不推荐将其用于生产环境中

4.应用层 这一层就是不同场景下对于Ceph各个应用接口的各种应用方式，例如基于librados直接开发的对象存储应用，基于RADOS GW开发的对象存储应用，基于RBD实现的云硬盘等等。

Ceph基本组件

- Osd 用于集群中所有数据与对象的存储。处理集群数据的复制、恢复、回填、再均衡。并向其他osd守护进程发送心跳，然后向Mon提供一些监控信息。
- Monitor 监控整个集群的状态，维护集群的cluster MAP二进制表，保证集群数据的一致性。ClusterMAP描述了对象块存储的物理位置，以及一个将设备聚合到物理位置的桶列表。
- MDS(可选) 为Ceph文件系统提供元数据计算、缓存与同步。在ceph中，元数据也是存储在osd节点中的，mds类似于元数据的代理缓存服务器。MDS进程并不是必须的进程，只有需要使用CEPHFS>时，才需要配置MDS节点。

CRUSH机制

引入CRUSH的目的

为了去中心化，避免单点故障，Ceph使用了CRUSH（Controlled Replication Under Scalable Hashing）算法，客户端根据它来计算数据被写到哪里去，以及从哪里读取所需数据。

理解CRUSH机制

对Ceph集群的一个读写操作，客户端首先访问Ceph monitor来获取cluster map 的一分副本，它包含五个map,分别是monitor map、OSD map 、 MDS map、CRUSH map 和PG map.

客户端通过这些cluster map知晓Ceph集群的状态和配置。通过CRUSH算法计算出或获取数据的主（primary）、次（secondary）和再次（tertiary）OSD的位置。

所有这些计算操作都是在客户端完成的，因此它们不会影响Ceph集群服务器端的性能。

每个map的简介如下：

- monitor map: 它包含监视节点端到端的信息，包括Ceph集群ID、monitor节点名称（hostname）、IP地址和端口号等。它还保存自monitor map被创建以来的最新版本号（epoch:每种map都维护着其历史版本，每个版本被称为一个epoch，epoch是一个单调递增的序号），以及最后修改时间等。
  查看monitor map 命令: ceph mon dump
- OSD map: 它保存一些常用的信息，包括集群ID、OSD map自创建以来的最新版本号（epoch）及其最后修改时间，以及存储池相关的信息，包括存储池名称、ID、类型、副本级别（replication level）和PG。它还保存着OSD的信息，比如数量、状态、权重、最后清理间隔（last clean interval）以及OSD节点的信息。
  查看OSD map命令：ceph osd dump
- PG map:它保存的信息包括PG的版本、时间戳、OSD map的最新版本号（epoch）、容量已满百分比，容量将满百分比等。它还记录了每个PG的ID、对象数量、状态、状态时间戳、up OSD sets、acting OSD sets,以及清理的信息。
  查看PG map 命令: ceph pg dump
- CRUSH map: 它保存的信息包括集群设备列表、bucket列表、故障域分层结构、故障域定义的规则等。
  查看CRUSH map 命令: ceph osd crush dump
- MDS map:它保存的信息包括MDS map当前版本号（epoch）、MDS map的创建和修改时间、数据和元数据存储池的ID、集群MDS数量以及MDS状态。
  查看MDS map 命令: ceph mds dump

CRUSH Map的内容

CRUSH算法通过计算数据存储位置来确定如何存储和检索。 CRUSH 授权 Ceph 客户端直接连接 OSD ，而非通过一个中央服务器或经纪人。数据存储、检索算法的使用，使 Ceph 避免了单点故障、性能瓶颈、和伸缩的物理限制。

CRUSH图包含 OSD 列表、把设备汇聚为物理位置的“桶”列表和指示 CRUSH 如何复制存储池里的数据的规则列表。

CRUSH图主要有 4 个主要段落：

1.设备
设备的格式：

```
#devices
device {num} {osd.name}
```

2.桶类型： 定义了 CRUSH 分级结构里要用的桶类型（ types ） 如：

```
# types
type 0 osd
type 1 host
type 2 chassis
type 3 rack
type 4 row
type 5 pdu
type 6 pod
type 7 room
type 8 datacenter
type 9 region
type 10 root
```

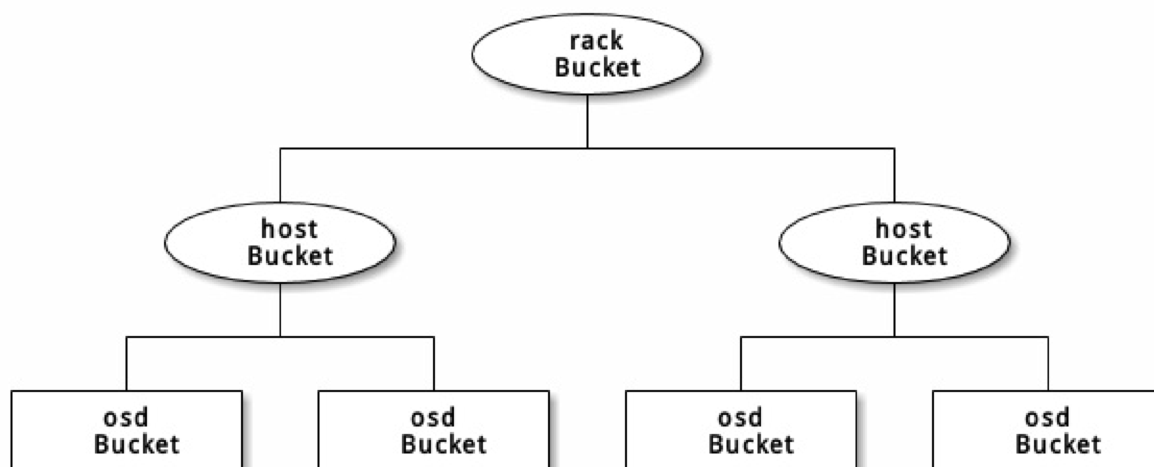
3.桶例程： 定义了桶类型后，还必须声明主机的桶类型、以及规划的其它故障域。
格式：

```
[bucket-type] [bucket-name] {
        id [a unique negative numeric ID]
        weight [the relative capacity/capability of the item(s)]
        alg [the bucket type: uniform | list | tree | straw ]
        hash [the hash type: 0 by default]
        item [item-name] weight [weight]
}
```

Ceph 支持四种桶，每种都是性能和组织简易间的折衷。如果你不确定用哪种桶，我们建议 straw 。关于桶类型的详细讨论请参考http://docs.ceph.org.cn/rados/operations/crush-map

各个桶都用了一种哈希算法，当前 Ceph 仅支持 rjenkins1 ，输入 0 表示哈希算法设置为 rjenkins1 。

例子如下：



定义的桶例程为：

```
host node1 {
        id -1
        alg straw
        hash 0
        item osd.0 weight 1.00
        item osd.1 weight 1.00
}

host node2 {
        id -2
        alg straw
        hash 0
        item osd.2 weight 1.00
        item osd.3 weight 1.00
}

rack rack1 {
        id -3
        alg straw
        hash 0
        item node1 weight 2.00
        item node2 weight 2.00
}
```

此例中，机柜桶不包含任何 OSD ，它只包含低一级的主机桶、以及其内条目的权重之和

4.规则： 由选择桶的方法组成。

规则格式如下：

```
rule <rulename> {

        ruleset <ruleset>
        type [ replicated | erasure ]
        min_size <min-size>
        max_size <max-size>
        step take <bucket-type>
        step [choose|chooseleaf] [firstn|indep] <N> <bucket-type>
        step emit
}
```

各字段含义如下:

4.1ruleset

描述: 区分一条规则属于某个规则集的手段。给存储池设置规则集后激活。
目的: 规则掩码的一个组件。
类型: Integer
是否必需: Yes
默认值: 0

4.2type

描述: 为硬盘（复制的）或 RAID 写一条规则。
目的: 规则掩码的一个组件。
类型: String
是否必需: Yes
默认值: replicated
合法取值: 当前仅支持 replicated 和 erasure

4.3min_size

描述: 如果一个归置组副本数小于此数， CRUSH 将不应用此规则。
类型: Integer
目的: 规则掩码的一个组件。
是否必需: Yes
默认值: 1

4.4max_size

描述: 如果一个归置组副本数大于此数， CRUSH 将不应用此规则。

类型: Integer

目的: 规则掩码的一个组件。

是否必需: Yes

默认值: 10

4.5step take

描述: 选取桶名并迭代到树底。

目的: 规则掩码的一个组件。

是否必需: Yes

实例: step take default

4.6step choose firstn {num} type {bucket-type}

描述: 选取指定类型桶的数量，这个数字通常是存储池的副本数（即 pool size ）。

如果 {num} == 0 选择 pool-num-replicas 个桶（所有可用的）；

如果 {num} > 0 && < pool-num-replicas 就选择那么多的桶；

如果 {num} < 0 它意为 pool-num-replicas - {num} 。

目的: 规则掩码的一个组件。

先决条件: 跟在 step take 或 step choose 之后。

实例: step choose firstn 1 type row

4.7step chooseleaf firstn {num} type {bucket-type}

描述: 选择 {bucket-type} 类型的一堆桶，并从各桶的子树里选择一个叶子节点。集合内桶的数量通常是存储池的副本数（即 pool size ）。

如果 {num} == 0 选择 pool-num-replicas 个桶（所有可用的）；

如果 {num} > 0 && < pool-num-replicas 就选择那么多的桶；

如果 {num} < 0 它意为 pool-num-replicas - {num} 。

目的: 规则掩码的一个组件。 它的使用避免了通过两步来选择一设备。

先决条件: Follows step take or step choose.

4.8step emit

描述: 输出当前值并清空堆栈。通常用于规则末尾，也适用于相同规则应用到不同树的情况。

目的: 规则掩码的一个组件。

先决条件: Follows step choose.

实例: step emit

较新版本的 CRUSH （从 0.48 起）为了解决一些遗留值导致几个不当行为，在最前面加入了一些参数值。

一个例子如下：

```
# begin crush map
tunable choose_local_tries 0 #本地重试次数。以前是 2 ，最优值是 0 。
tunable choose_local_fallback_tries 0 #以前 5 ，现在是 0
tunable choose_total_tries 50 #选择一个条目的最大尝试次数。以前 19 ，后来
tunable chooseleaf_descend_once 1 #是否重递归叶子选择，或只试一次、并允许
tunable straw_calc_version 1

# devices
device 0 osd.0
device 1 osd.1
device 2 osd.2

# types
type 0 osd
type 1 host
type 2 chassis
type 3 rack
type 4 row
type 5 pdu
type 6 pod
type 7 room
type 8 datacenter
type 9 region
type 10 root

# buckets
host server-250 {
        id -2           # do not change unnecessarily
        # weight 2.160
        alg straw
        hash 0  # rjenkins1
        item osd.0 weight 0.720
```

```
        item osd.1 weight 0.720
        item osd.2 weight 0.720
}
root default {
        id -1           # do not change unnecessarily
        # weight 2.160
        alg straw
        hash 0  # rjenkins1
        item server-250 weight 2.160
}

# rules
rule replicated_ruleset {
        ruleset 0
        type replicated
        min_size 1
        max_size 10
        step take default
        step chooseleaf firstn 0 type osd
        step emit
}

# end crush map
```

如何编辑CRUSH map

1.从任意一个monitor节点上获取CRUSH map

```
ceph osd getcrushmap -o crushmap
```

2.反编译它，让它成为我们能阅读的格式

```
crushtool -d crushmap -o crushmap.txt
```

3.修改相应的内容

```
vim  crushmap.txt
```

4.重新编译

```
crushtool -c crushmap.txt -o newcrushmap
```

5.将重新编译的CRUSH map注入Ceph集群

```
ceph osd setcrushmap -i newcrushmap
```

puppet-ceph部署

Ceph集群有多种部署方式，诸如Ansible、Puppet和Chef等配置管理工具都可以按照你喜欢的方式来安装和部署Ceph集群。这里我们只介绍Puppet的部署方式。

准备工作

在puppet master module目录下下载puppet-ceph模块，下载地址为https://github.com/openstack/puppet-ceph/tree/stable/hammer

openstack/puppet-ceph 使用ceph版本为hammer

此例中，我们部署一个mon节点，两个osd节点，hostname分别为：test-ceph-1,test-ceph-2,test-ceph-3

部署指南

各节点加载的类：

```
node /^test-ceph-1$/ {
$ceph_pools = ['test']
ceph::pool { $ceph_pools: }
class { '::ceph::profile::mon': }
}
node /^test-ceph-[2-3]$/ {
class { '::ceph::profile::osd': }
}
```

传的hieradata:

common/ceph.yaml:

```
---
######## Ceph
ceph::profile::params::release: 'hammer'

######## Ceph.conf
ceph::profile::params::fsid: '4b5c8c0a-ff60-454b-a1b4-9747aa737d19
ceph::profile::params::authentication_type: 'cephx'
ceph::profile::params::mon_initial_members: 'test-ceph-1'
ceph::profile::params::mon_host: '10.0.86.23:6789'
ceph::profile::params::osd_pool_default_size: '2'

######## Keys
ceph::profile::params::mon_key: 'AQATGHJTUCBqIBAA7M2yafV1xctn1pgr3C
ceph::profile::params::client_keys:
  'client.admin':
    secret: 'AQATGHJTUCBqIBAA7M2yafV1xctn1pgr3GcKPg=='
    mode: '0600'
    cap_mon: 'allow *'
    cap_osd: 'allow *'
    cap_mds: 'allow *'
  'client.bootstrap-osd':
    secret: 'AQATGHJTUCBqIBAA7M2yafV1xctn1pgr3GcKPg=='
    keyring_path: '/var/lib/ceph/bootstrap-osd/ceph.keyring'
    cap_mon: 'allow profile bootstrap-osd'
```

test-ceph-2.yaml:

```
ceph::profile::params::osds:
  '/dev/sdb':
    journal: ''
```

test-ceph-3.yaml:

```
ceph::profile::params::osds:
  '/dev/sdc':
    journal: ''
```

参数说明

puppet会执行 ceph-disk prepare /dev/sdc ，如果journal为空，它会把自动把这块盘分成两个分区,一个为ceph data ,一个为ceph journal。journal分区大小默认为5G，剩下的 都分给ceph data.

Journal的作用类似于mysql innodb引擎中的事物日志系统。当有突发的大量写入操作时，ceph可以先把一些零散的，随机的IO请求保存到缓存中进行合并，然后再统一向内核发起IO请求。journal的io是非常密集的,很大程度上也损耗了硬件的io性能，所以通常在生产环境中，推荐使用ssd来单独存储journal文件以提高ceph读写性能。

journal也可以使用单独的数据盘，只需要在hieradata中传递相应的设备名即可。

openstack/puppet-ceph 传osds参数不支持wwn的方式,因为ceph-disk当前不支持使用wwn来作为磁盘标识的输入参数。

如果重启了mon节点，需要执行：

```
service ceph start mon.server-250
```

如果重启了osd节点，需要执行：

```
ceph-disk activate-all
```

activate-all 靠 /dev/disk/by-parttype-uuid/$typeuuid.$uuid 发现所有分区

parttype-uuid 是在执行activate-prepare 时生成的。通过parttypeuuid，在本机插拔osd盘完全不会导致故障。

puppet执行过程分析

创建mon的大致过程如下：

1.安装包

```
package { $::ceph::params::packages :
    ensure => $ensure,
    tag    => 'ceph'
  }
```

.是否开启认证

```
# [*authentication_type*] Activate or deactivate authentication
#   Optional. Default to cephx.
#   Authentication is activated if the value is 'cephx' and deactiv
#   if the value is 'none'. If the value is 'cephx', at least one o
#   key or keyring must be provided.
if $authentication_type == 'cephx' {
      ceph_config {
        'global/auth_cluster_required': value => 'cephx';
        'global/auth_service_required': value => 'cephx';
        'global/auth_client_required':  value => 'cephx';
        'global/auth_supported':        value => 'cephx';
      }
```

3.生成mon密钥

```
cat > ${keyring_path} << EOF
[mon.]
key = ${key}
caps mon = \"allow *\"
EOF
chmod 0444 ${keyring_path}
```

4.生成/etc/ceph/ceph.client.admin.keyring文件

```
touch /etc/ceph/${cluster_name}.client.admin.keyring
```

5.初始化monitor服务，创建done,sysvinit空文件

```
mon_data=\$(ceph-mon ${cluster_option} --id ${id} --show-config-val
if [ ! -d \$mon_data ] ; then
  mkdir -p \$mon_data
  if ceph-mon ${cluster_option} \
        --mkfs \
        --id ${id} \
        --keyring ${keyring_path} ; then
    touch \$mon_data/done \$mon_data/${init} \$mon_data/keyring
  else
    rm -fr \$mon_data
  fi
fi
```

6.启动mon服务：

```
service ceph start mon.test-ceph-xue-1
```

建osd的大致过程如下：

1.安装包

```
package { $::ceph::params::packages :
    ensure => $ensure,
    tag    => 'ceph'
  }
```

2.是否开启认证

```
# [*authentication_type*] Activate or deactivate authentication
#   Optional. Default to cephx.
#   Authentication is activated if the value is 'cephx' and deacti
#   if the value is 'none'. If the value is 'cephx', at least one 
#   key or keyring must be provided.
if $authentication_type == 'cephx' {
      ceph_config {
        'global/auth_cluster_required': value => 'cephx';
        'global/auth_service_required': value => 'cephx';
        'global/auth_client_required':  value => 'cephx';
        'global/auth_supported':        value => 'cephx';
      }
```

3.创建keyring file

```
if ! defined(File[$keyring_path]) {
    file { $keyring_path:
      ensure  => file,
      owner   => $user,
      group   => $group,
      mode    => $mode,
      require => Package['ceph'],
    }
  }
```

4.生成管理员密钥环，生成 client.admin 用户并加入密钥环

```
ceph-authtool \$NEW_KEYRING --name '${name}' --add-key '${secret}'
```

5.把 client.admin 密钥加入 ceph.mon.keyring

```
ceph ${cluster_option} ${inject_id_option} ${inject_keyring_option]
```

6.ceph 0.94版本下禁用udev rules,否则可能会导致ceph-disk activate失败

```
mv -f ${udev_rules_file} ${udev_rules_file}.disabled && udevadm cor
```

7.使用ceph-disk prepare 做预处理 预处理用作 Ceph OSD 的目录、磁盘。它会创建 GPT 分区、给分区打上 Ceph 风格的 uuid 标记、创建文件系统、把此文件系统标记为已就绪、使用日志磁盘的整个分区并新增一>分区。可单独使用，也可由 ceph-deploy 用。

```
if ! test -b ${data} ; then
mkdir -p ${data}
fi
ceph-disk prepare ${cluster_option} ${data} ${journal}
udevadm settle
```

8.激活 Ceph OSD 激活 Ceph OSD 。先把此卷挂载到一临时位置，分配 OSD 惟一标识符（若有必要），重挂载到正确位置

```
ceph-disk activate ${data}
```

小结

在这里，我们介绍了Ceph的一些基础知识和puppet-ceph搭建过程，本篇文章没有涉及到但是需要我们注意的是，在搭建ceph集群之前，要先对集群的硬件进行合理的规划，包括故障域和潜在的性能问题。

动手练习

1. 通过puppet部署了ceph集群之后，在本机插拔osd盘，再执行ceph-disk activate-all命令，观察ceph 集群状态
2. 导出ceph 集群的crush map图，理解其含义。
3. 学习并使用rbd命令创建、显示、对照（introspect）和移除块设备镜像。

第四章 Puppet-OpenStack模块

PuppetOpenstack项目发展到今日，代码经历了多次的迭代和持续的更新，其代码和规范可以称之为Puppet进阶的经典素材。它体现在以下几点：

- 严格遵守Puppet Code Style
- 松耦合的代码逻辑
- 几乎没有代码冗余，非常高的代码复用率
- 精心编写的自定义Resource Type和Facter，在灵活性和控制能力上做出了权衡

目前官方提供的模块有以下：

- Alarming (Aodh)
- Key Manager (Barbican)
- Telemetry (Ceilometer)
- Block Storage (Cinder)
- DNS (Designate)
- Image service (Glance)
- Time Series Database (Gnocchi)
- Orchestration (Heat)
- Dashboard (Horizon)
- Bare Metal (Ironic)
- Identity (Keystone)
- Shared Filesystems (Manila)
- Workflow service (Mistral)
- Application catalog (Murano)
- Networking (Neutron)
- Compute (Nova)
- Load Balancer (Octavia)
- Oslo libraries (Oslo)
- Benchmarking (Rally)
- Data processing (Sahara)
- Object Storage (Swift)
- Testing (Tempest)
- Deployment (TripleO)

- Database service (Trove)
- Deployment UI (TripleO UI)
- Root Cause Analysis (Vitrage)
- Message service (Zaqar)

本书将会覆盖核心Openstack服务和部分热门服务：

- Telemetry (Ceilometer)
- Block Storage (Cinder)
- Image service (Glance)
- Time Series Database (Gnocchi)
- Dashboard (Horizon)
- Identity (Keystone)
- Application catalog (Murano)
- Networking (Neutron)
- Compute (Nova)
- Object Storage (Swift)
- Testing (Tempest)
- Ceph(Block Storage)
- Benchmarking (Rally)
- Data processing (Sahara)
- Orchestration (Heat)
- DNS (Designate)

- 第四章 Puppet-OpenStack模块

OpenStack模块代码结构

1.简介

在开始介绍各个OpenStack服务的Puppet模块前，先观察一下所有OpenStack module的目录结构，你会发现所有的模块的部分代码目录结构和命名方式几乎是一致的，这是经过了长期迭代和开发中形成的规范和统一，代码结构统一带来的好处有两点：

1. 易于维护人员理解和管理
2. 减少冗余代码，提高代码复用

那么我们就来看看一个OpenStack服务的Module中包含了哪些目录：

- examples/ 放置示例代码
- ext/ 放置external代码，和主要代码无关，但是一些有用的脚本
- lib/ 放置library代码，例如自定义facter,resource type
- manifests/ 放置puppet代码
- releasenotes/ 放置releasenote
- spec/ 放置class,unit,acceptance测试
- tests/ 已弃用，使用examples替代

以上目录中最重要的是manifests目录，用于放置Puppet代码，在该目录下包含了以下通用代码文件：

名称	说明
init.pp	主类，也称为入口类，通常仅用于管理公共参数（如MQ参数）
params.pp	用于特定操作系统的参数值设置
client.pp	管理客户端的配置
config.pp	用于管理自定义的参数配置
policy.pp	policy设置
db/	支持多种数据库后端的配置
keystone/	keystone endpoint,service,user,role的设置

2.数据库管理

2.1 `class <service>::db`

`class <service>::db` 用于管理各OpenStack服务中的数据库相关配置， `<service>` 是OpenStack服务的名称，以Aodh为例：

```
class aodh::db (
  $database_db_max_retries = $::os_service_default,
  $database_connection     = 'sqlite:////var/lib/aodh/aodh.sqlite',
  $database_idle_timeout   = $::os_service_default,
  $database_min_pool_size  = $::os_service_default,
  $database_max_pool_size  = $::os_service_default,
  $database_max_retries    = $::os_service_default,
  $database_retry_interval = $::os_service_default,
  $database_max_overflow   = $::os_service_default,
) {

  include ::aodh::deps

  $database_connection_real = pick($::aodh::database_connection, $d
  $database_idle_timeout_real = pick($::aodh::database_idle_timeout
  $database_min_pool_size_real = pick($::aodh::database_min_pool_si
  $database_max_pool_size_real = pick($::aodh::database_max_pool_si
  $database_max_retries_real = pick($::aodh::database_max_retries,
  $database_retry_interval_real = pick($::aodh::database_retry_inte
  $database_max_overflow_real = pick($::aodh::database_max_overflow

  oslo::db { 'aodh_config':
    db_max_retries => $database_db_max_retries,
    connection     => $database_connection_real,
    idle_timeout   => $database_idle_timeout_real,
    min_pool_size  => $database_min_pool_size_real,
    max_pool_size  => $database_max_pool_size_real,
    max_retries    => $database_max_retries_real,
    retry_interval => $database_retry_interval_real,
    max_overflow   => $database_max_overflow_real,
  }
}
```

`class aodh::db` 管理了与数据库相关的配置项，其中通过调用 `oslo::db` 来实现，关于 `puppet-oslo` 模块，本书会在下一章节详细说明。

2.2 `class <service>::db::mysql`

`class <service>::db::mysql` 用于创建相关服务的MySQL数据库，用户和授权等。以Aodh为例:

```
class aodh::db::mysql(
  $password,
  $dbname        = 'aodh',
  $user          = 'aodh',
  $host          = '127.0.0.1',
  $charset       = 'utf8',
  $collate       = 'utf8_general_ci',
  $allowed_hosts = undef
) {

  include ::aodh::deps

  validate_string($password)

  ::openstacklib::db::mysql { 'aodh':
    user          => $user,
    password_hash => mysql_password($password),
    dbname        => $dbname,
    host          => $host,
    charset       => $charset,
    collate       => $collate,
    allowed_hosts => $allowed_hosts,
  }

  Anchor['aodh::db::begin']
  ~> Class['aodh::db::mysql']
  ~> Anchor['aodh::db::end']

}
```

`class aodh::db::mysql` 管理了MySQL aodh数据库的创建，aodh用户创建和密码设定，数据库编码，访问授权等。其调用了openstacklib::db::mysql来实现上述功能，关于 `puppet-openstacklib` 模块，本书会在下一章节详细说明。

2.3 `class <service>::db::postgresql`

`class <service>::db::mysql` 用于创建相关服务的PostgreSQL数据库，用户和授权等。以Aodh为例:

```
class aodh::db::postgresql(
  $password,
  $dbname     = 'aodh',
  $user       = 'aodh',
  $encoding   = undef,
  $privileges = 'ALL',
) {

  include ::aodh::deps

  ::openstacklib::db::postgresql { 'aodh':
    password_hash => postgresql_password($user, $password),
    dbname        => $dbname,
    user          => $user,
    encoding      => $encoding,
    privileges    => $privileges,
  }

  Anchor['aodh::db::begin']
  ~> Class['aodh::db::postgresql']
  ~> Anchor['aodh::db::end']

}
```

`class aodh::db::postgresql` 完成了aodh数据库的创建，aodh用户创建和密码设定，数据库编码，访问授权等。其调用了openstacklib::db::postgresql来实现上述功能。

2.4 `class <service>::db::sync`

`class aodh::db::sync` 用于执行数据库表的初始化和更新操作。以Aodh为例:

```
class aodh::db::sync (
  $user = 'aodh',
){

  include ::aodh::deps

  exec { 'aodh-db-sync':
    command     => 'aodh-dbsync --config-file /etc/aodh/aodh.conf',
    path        => '/usr/bin',
    refreshonly => true,
    user        => $user,
    try_sleep   => 5,
    tries       => 10,
    logoutput   => on_failure,
    subscribe   => [
      Anchor['aodh::install::end'],
      Anchor['aodh::config::end'],
      Anchor['aodh::dbsync::begin']
    ],
    notify      => Anchor['aodh::dbsync::end'],
  }

}
```

aodh::db::sync的实现是通过声明exec资源来调用aodh-dbsync命令行完成数据库初始化的操作。

3. Keystone初始化管理

在OpenStack部署工作中，与Keystone相关的初始化操作是集群正常运行必不可少的步骤：

- 创建Domain
- 创建Project
- 创建User，设置Password
- 创建并指定Role
- 创建Service

- 创建Endpoint

也包括在后期的运维中，指定user的password更新或者endpoint的更改等常见操作都可以在Puppet中完成。而这背后的工作是通过 `<service>::keystone::auth` 来完成的。

3.1 `class <service>::keystone::auth`

`<service>::keystone::auth` 用于创建OpenStack服务的user,service和endpoint，以Aodh为例：

```
class aodh::keystone::auth (
  $password,
  $auth_name              = 'aodh',
  $email                  = 'aodh@localhost',
  $tenant                 = 'services',
  $configure_endpoint   = true,
  $configure_user        = true,
  $configure_user_role = true,
  $service_name          = 'aodh',
  $service_type          = 'alarming',
  $region                 = 'RegionOne',
  $public_url            = 'http://127.0.0.1:8042',
  $internal_url          = 'http://127.0.0.1:8042',
  $admin_url              = 'http://127.0.0.1:8042',
) {

  include ::aodh::deps

  keystone::resource::service_identity { 'aodh':
    configure_user         => $configure_user,
    configure_user_role  => $configure_user_role,
    configure_endpoint    => $configure_endpoint,
    service_name           => $service_name,
    service_type           => $service_type,
    service_description  => 'OpenStack Alarming Service',
    region                  => $region,
    auth_name               => $auth_name,
    password                => $password,
    email                   => $email,
    tenant                  => $tenant,
    public_url             => $public_url,
    internal_url           => $internal_url,
    admin_url               => $admin_url,
  }

}
```

实际上 `aodh::keystone::auth` 在声明 `define keystone::resource::service_identity` 的基础上，根据Aodh服务而重写了相关的参数。

下面来看一段代码，关于 `keystone::resource::service_identity` 如何实现service的管理：

```
if $configure_service {
  if $service_type {
    ensure_resource('keystone_service', "${service_name_real}::$
      'ensure'      => $ensure,
      'description' => $service_description,
    })
  } else {
    fail ('When configuring a service, you need to set the servic
  }
}
```

通过函数 `ensure_resource` 调用了 `keystone_service` 自定义资源类型，并传入两个参数:

- "${service_name_real}::${service_type}"
- {'ensure' => $ensure, 'description' => $service_description,}

有细心的读者读到这里可能会好奇，把服务名称和服务类型作为一个参数传入keystone_service，它是怎么区分的？

先来看keystone_service.rb的代码片段(代码路径puppet-keystone/lib/puppet/type/keystone_service.rb)：

```
def self.title_patterns
  PuppetX::Keystone::CompositeNamevar.basic_split_title_patterns(
end
```

title_patterns方法通过调用 `PuppetX::Keystone::CompositeNamevar.basic_split_title_patterns` 方法来得到 `:name` 和 `:type` 变量。

接着跳转到basic_split_title_patterns的定义(代码路径lib/puppet_x/keystone/composite_namevar.rb):

```
def self.not_two_colon_regex
  # Anything but 2 consecutive colons.
  Regexp.new(/(?:[^:]|:[^:])+/)
end

def self.basic_split_title_patterns(prefix, suffix, separator =
  associated_regexps = []
  if regexps.empty? and separator == '::'
    associated_regexps += [not_two_colon_regex, not_two_colon_reg
  else
    if regexps.count != 2
      raise(Puppet::DevError, 'You must provide two regexps')
    else
      associated_regexps += regexps
    end
  end
  prefix_re = associated_regexps[0]
  suffix_re = associated_regexps[1]
  [
    [
      /^(#{prefix_re})#{separator}(#{suffix_re})$/,
      [
        [prefix],
        [suffix]
      ]
    ],
    [
      /^(#{prefix_re})$/,
      [
        [prefix]
      ]
    ]
  ]
end
```

可以看到 `basic_split_title_patterns` 方法默认使用'::'作为分隔符，通过not_two_colon_regex函数进行正则匹配并切割字符串。 至此，我们从上到下地剖析了如何实现Keystone相关资源的初始化，以加深读者对于代码的理解。在实际使用中，对于终端用户来说，并不需要关心底层的Ruby代码。

3.2 `class <service>::keystone::authtoken`

`<service>::keystone::authtoken` 用于管理OpenStack各服务配置文件中的keystone_authtoken配置节。以Aodh服务为例:

```
class aodh::keystone::authtoken(
...){

  ...

  keystone::resource::authtoken { 'aodh_config':
    username                     => $username,
    password                     => $password,
    project_name                 => $project_name,
    auth_url                     => $auth_url,
    auth_uri                     => $auth_uri,
    auth_version                 => $auth_version,
    auth_type                    => $auth_type,
    auth_section                 => $auth_section,
    ...
    memcache_pool_conn_get_timeout => $memcache_pool_conn_get_time
    memcache_pool_dead_retry     => $memcache_pool_dead_retry,
    memcache_pool_maxsize        => $memcache_pool_maxsize,
    memcache_pool_socket_timeout   => $memcache_pool_socket_timeout
    ...
  }
}
```

`aodh::keystone::authtoken` 定义中声明了 `define keystone::resource::authtoken` ，并重写了部分参数的默认值。 `keystone::resource::authtoken` 中定义了hash类型变量

$keystonemiddleware_options，涵盖了keystone_authtoken配置节下的所有参数，最终通过调用create_resources函数，传入服务名称参数$name，从而完成指定服务配置文件中keystone_authtoken的配置。

```
$keystonemiddleware_options = {
  'keystone_authtoken/auth_section'                 => {'value
  'keystone_authtoken/auth_uri'                     => {'value
  'keystone_authtoken/auth_type'                    => {'value
  'keystone_authtoken/auth_version'                 => {'value
  'keystone_authtoken/cache'                        => {'value
   ...
  'keystone_authtoken/username'                     => {'value
  'keystone_authtoken/password'                     => {'value
  'keystone_authtoken/user_domain_name'             => {'value
  'keystone_authtoken/project_name'                 => {'value
  'keystone_authtoken/project_domain_name'          => {'value
  'keystone_authtoken/insecure'                     => {'value
}
create_resources($name, $keystonemiddleware_options)
```

4.维护不同Linux发行版之间的数据

PuppetOpenstack支持在Redhat, CentOS, Ubuntu等多个Linux发行版上部署OpenStack服务，然而在不同的Linux发行版中，同一个OpenStack服务的软件包的名称会有所不同。

例如，Nova API软件包的名称在Redhat下是'openstack-nova-api'，在Debian下是'nova-api'。

而这些数据则通过各个模块的 `class <service>::params` 维护。

以keystone::params为例，可以看到不同的Linux发行版之间$package_name, $service_name等参数值也有所不同:

```
class keystone::params {
  include ::openstacklib::defaults
  $client_package_name = 'python-keystoneclient'
  $keystone_user       = 'keystone'
  $keystone_group      = 'keystone'
  $keystone_wsgi_admin_script_path  = '/usr/bin/keystone-wsgi-admin
  $keystone_wsgi_public_script_path = '/usr/bin/keystone-wsgi-publi
  case $::osfamily {
    'Debian': {
      $package_name                 = 'keystone'
      $service_name                 = 'keystone'
      $keystone_wsgi_script_path    = '/usr/lib/cgi-bin/keystone'
      $python_memcache_package_name = 'python-memcache'
      $mellon_package_name          = 'libapache2-mod-auth-mellon'
      $openidc_package_name         = 'libapache2-mod-auth-openidc
    }
    'RedHat': {
      $package_name                 = 'openstack-keystone'
      $service_name                 = 'openstack-keystone'
      $keystone_wsgi_script_path    = '/var/www/cgi-bin/keystone'
      $python_memcache_package_name = 'python-memcached'
      $mellon_package_name          = 'mod_auth_mellon'
      $openidc_package_name         = 'mod_auth_openidc'
    }
    default: {
      fail("Unsupported osfamily ${::osfamily}")
    }
  }
}
```

5. 管理自定义配置项的 `<service>::config`

模板是用于管理配置文件的常见方式，对于成熟的项目而言，模板是一种理想的管理配置文件方式。但对于快速迭代的项目如OpenStack，维护人员会非常痛苦，每增删一个配置项需要同时更新模板和manifets文件。

试想一个module的更新若都在参数的增添上，那对社区开发者来说是极大的成本。有没有一种办法可以不修改module，直接在hiera里定义来添加新配置项呢？

`<service>::config` 类是由笔者在14年初提出的特性，目的是灵活地管理自定义配置项。

自定义配置项是指未被模块管理的参数。怎么理解？

以 `keystone::config` 为例，其核心是create_resources函数以及 `keystone_config/keystone_paste_init` 自定义资源：

```
# == Class: keystone::config
#
# This class is used to manage arbitrary keystone configurations.
#
# === Parameters
#
# [*keystone_config*]
#   (optional) Allow configuration of arbitrary keystone configurat
#   The value is an hash of keystone_config resources. Example:
#   { 'DEFAULT/foo' => { value => 'fooValue'},
#     'DEFAULT/bar' => { value => 'barValue'}
#   }
#   In yaml format, Example:
#   keystone_config:
#     DEFAULT/foo:
#       value: fooValue
#     DEFAULT/bar:
#       value: barValue
#
# [*keystone_paste_ini*]
#   (optional) Allow configuration of /etc/keystone/keystone-paste.
#
#   NOTE: The configuration MUST NOT be already handled by this moc
#   or Puppet catalog compilation will fail with duplicate resource
#
class keystone::config (
  $keystone_config    = {},
  $keystone_paste_ini = {},
) {

  include ::keystone::deps

  validate_hash($keystone_config)
  validate_hash($keystone_paste_ini)

  create_resources('keystone_config', $keystone_config)
  create_resources('keystone_paste_ini', $keystone_paste_ini)
}
```

若Keystone在某版本新增了参数new_param，在puppet-keystone模块里没有该参数，此时，只要使用keystone::config就可以轻松完成参数的管理。

在hiera文件中添加以下代码：

```
---
   keystone::config::keystone_config:
     DEFAULT/new_param:
       value: newValue
```

6.管理客户端 `<service>::client`

`<service>::client` 用于管理各OpenStack服务的Client端，完成客户端的安装。

以Nova为例，nova::client完成了 `python-novaclient` 软件包的安装：

```
class nova::client(
  $ensure = 'present'
) {
  include ::nova::deps

  package { 'python-novaclient':
    ensure => $ensure,
    tag    => ['openstack', 'nova-support-package'],
  }

}
```

7. 管理策略 `<service>::policy`

`<service>::policy` 用于管理Openstack各服务的策略文件policy.json。

以Cinder为例，下面是cinder::policy代码：

```
class cinder::policy (
  $policies    = {},
  $policy_path = '/etc/cinder/policy.json',
) {

  include ::cinder::deps

  validate_hash($policies)

  Openstacklib::Policy::Base {
    file_path => $policy_path,
  }

  create_resources('openstacklib::policy::base', $policies)
  oslo::policy { 'cinder_config': policy_file => $policy_path }

}
```

其中使用create_resources调用了`openstacklib::policy::base`，以及声明了oslo::policy定义。

`puppet-keystone` 模块介绍

0.基础知识

`puppet-keystone` 是用于配置和管理Keystone，其中包括:服务，软件包，Keystone user，role，service，endpoint等等。其中keystone user, role, service, endpoint等资源的管理是通过自定义的resource type来实现。

在开始介绍puppet-keystone模块前，先来回顾一下Keystone中的基础概念。

Identity

Keystone的Identity： `user` 和 `group` ，用于标识用户的身份，数据可以存在Keystone数据库中，或者也可以使用LDAP。

名称	说明
user	user表示独立的API消费者，user非全局唯一，必须属于某个domain，但在domain命名空间下唯一
group	group表示汇总user集合的容器，和user一样，group非全局唯一，必须属于某个domain，在domain命名空间下唯一

Resources

Keystone的resources部分提供了两类数据：`Projects` 和 `Domains` ，通常存储在SQL中。

名称	说明
Project(Tenant)	Project(在v2.0时也称为Tenant)表示Openstack基本单位的所有权限。在OpenStack中的资源必须归属于某个特定project。project非全局唯一，必须归属于某个domain，在domain命名空间下唯一。若一个project没有被指定domain，那么其domain会被设置为default
Domain	domain是project，user和group更高层级的容器。每个domain定义了一个命名空间，Keystone默认提供了一个名为'Default'的默认domain。Domain是全局唯一的。

Assignment

Assignment提供了role和role assignment的数据。

名称	说明
Role	role指定了user能获取的授权级别，roles可以domain或project级别授予，role可以被指定到单独的user或group。注意噢，role可是全局唯一的。
Role Assignments	一个包含Role, Resource, Identity的三元组

Token

Token服务用于验证和管理token，在完成对用户正确的认证请求后，Keystone会返回相应的token，token存在有效期。在用户与Openstack服务的交互中，会使用token作为验证信息，提高系统的安全性。

Catalog

Catalog提供了各个service的endpoint注册入口，用于endpoint自动发现。

以下是Keystone service catalog的样例：

```
"catalog": [
    {
        "name": "Keystone",
        "type": "identity",
        "endpoints": [
            {
                "interface": "public",
                "url": "https://identity.example.com:35357/"
            }
        ]
    }
]
```

通常，作为用户不需要关心这个列表，catalog在以下情况下会作为返回值响应：

- token creation response (`POST /v3/auth/tokens`)
- token validation response (`GET /v3/auth/tokens`)
- standalone resource (`GET /v3/auth/catalog`)

Services

service catalog本身是由一组services组成，service的定义是：

> Service实体表示Openstack中的web服务。每个service可以有0个或以上的endpoint，当然没有endpoint的service并没有什么实际用途。完整描述请参见：Identity API v3 spec

除了和endpoint相关以外，还有两个非常重要的属性：

- name (string)

> 面向用户的service名称

这表示该参数的值不是为了让程序去解析的，而是作为一个终端用户可读的字符串。例如keystone服务的name，你可以设置为"Keystone"或者"New Public Cloud Indetity Service"。因此，使用者可以根据实际需求来设置。

- type (string)

> 描述service所实现的API。该参数值只能在给定的列表中选择。目前Openstack支持的参数值有：`compute, image, ec2, identity, volume, network`等。

Endpoints

Endpoint表示API服务的基础URL，以及与其相关的metadata。每个服务应该有1个及以上相关的endpoint，例如：publicurl,adminurl,internalurl。

> Endpoint实体表示Opestack web services的URL。

- interface(string)

根据设置的类型来决定endpoint的访问权限：

- `public`：向终端用户提供可在公网上访问的网络接口
- `internal`：向终端用户提供近可在内部网络访问的网络接口
- `admin`：提供各个服务管理权限的访问，一般仅部署在内部并且加密的网络接口

多数服务在实际使用时，只需要设置`public`URL即可。

- url (string)

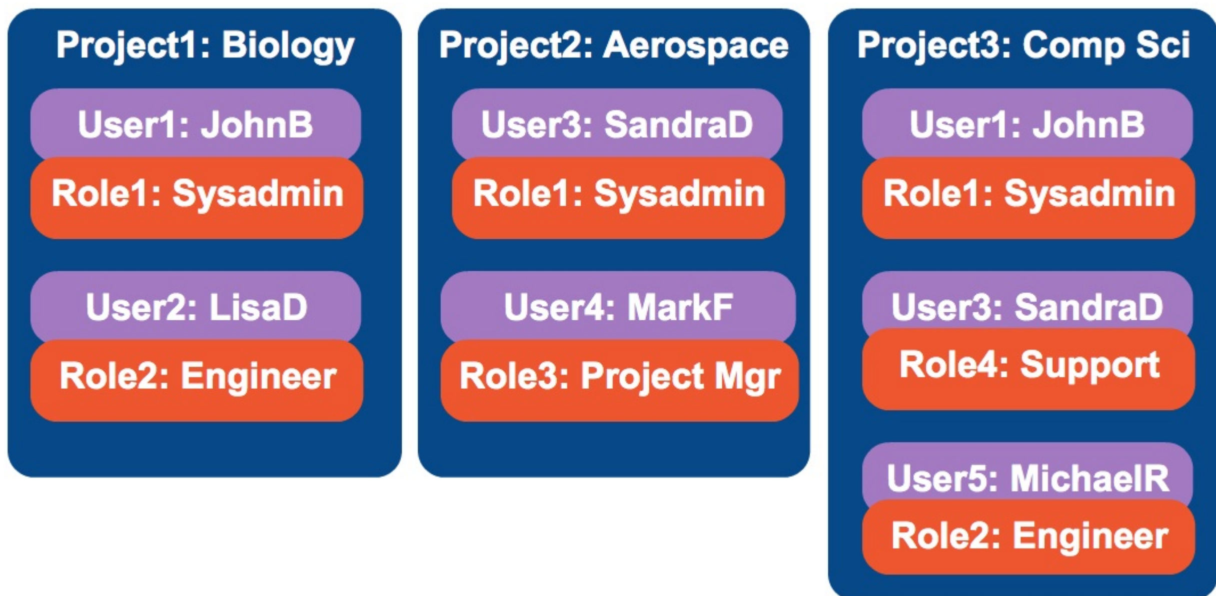
> service enpoint的完整URL。

这个完整URL应该由不带版本信息的基础URL加端口号组成。一个理想的url是：`https://identity.example.com:35357/`

相反,`https://identity.example.com:35357/v2.0/`作为一个反例，它引导所有的client去连接指定的v2.0版本，不管这些客户端能否处理哪里版本。

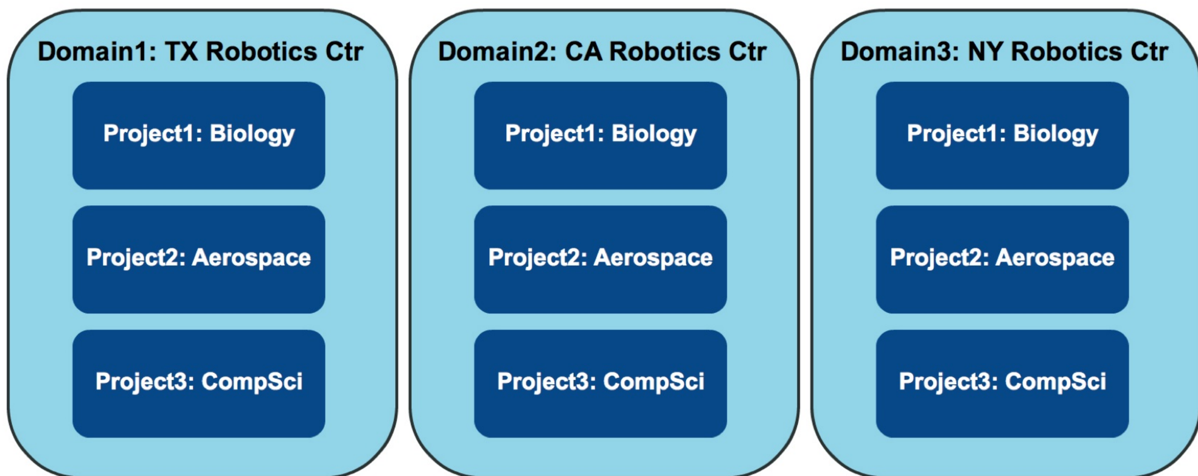
我们通过图例来解释这些复杂的概念：

Keystone v2 model



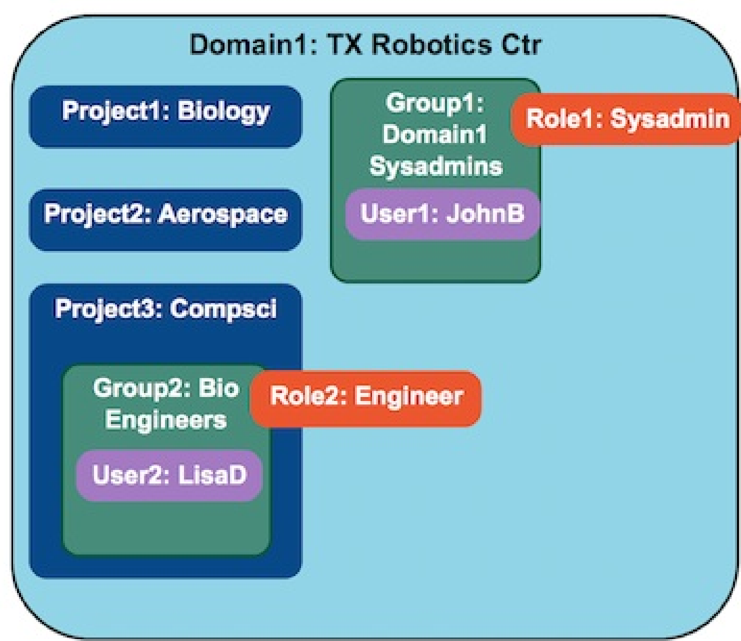
- user可以存在于不同的部门中（project），并且在各个部门中可以拥有不同的role。
- SandraD在Aerospace是个系统管理员，在Comp Sci就变身为客户支持。

Keystone v3 model: Domain



- v3通过domain术语引入了多租户的概念。如上图，domain相当于是project的容器。
- 通过domain，一个云用户就可以创建属于自己的user，groups和roles。

Keystone v3 model: Group



- 往常我们需要为user/project赋予role，现在domain owner就可以把role赋予group，然后把user添加到group里去。
- role可以赋予到domain范围的group或者project范围的group

在上图中：

- JohnB属于"domain1 sysadmins" group，拥有sysadmin role，并属于Bio,Aero,Compsci project。
- LisaD属于"Big Engineers"group，拥有Engineer role，仅属于compsci project。

Keystone服务组件

组件	描述
openstack-keystone	对外提供认证和授权服务，同时支持v2/v3 API
keystone	基于命令行的keystone客户端工具

1.先睹为快

不想看下面大段的代码解析，已经跃跃欲试了？

OK，我们开始吧！

打开虚拟机终端并输入以下命令：

```
$ puppet apply -v keystone/examples/v3_basic.pp
```

等待命令执行完成，在终端下试试吧：

```
    $ export OS_IDENTITY_API_VERSION=3
    $ export OS_USERNAME=admin
    $ export OS_USER_DOMAIN_NAME=admin_domain
    $ export OS_PASSWORD=ChangeMe
    $ export OS_PROJECT_NAME=admin
    $ export OS_PROJECT_DOMAIN_NAME=admin_domain
    $ export OS_AUTH_URL=http://keystone.local:35357/v3

    $ openstack user list
    $ openstack service list
```

这是如何做到的？下面来看v3_basic.pp代码

```
#设置了全局的Exec属性，当命令执行失败时，输出结果
Exec { logoutput => 'on_failure' }

# 安装MySQL服务
class { '::mysql::server': }
# 配置keystone database
class { '::keystone::db::mysql':
  password => 'keystone',
}
# 配置keystone服务
class { '::keystone':
  verbose             => true,
  debug               => true,
  database_connection => 'mysql://keystone:keystone@127.0.0.1/keyst
  admin_token         => 'admin_token',
  enabled             => true,
}
# 设置admin role
class { '::keystone::roles::admin':
  email               => 'test@example.tld',
  password            => 'a_big_secret',
  admin               => 'admin', # username
  admin_tenant        => 'admin', # project name
  admin_user_domain   => 'admin', # domain for user
  admin_tenant_domain => 'admin', # domain for project
}
# 创建keystone endpoint
class { '::keystone::endpoint':
  public_url => 'http://127.0.0.1:5000/',
  admin_url  => 'http://127.0.0.1:35357/',
}
```

2.核心代码讲解

2.1 class keystone

`class keystone` 逻辑非常复杂，暂先抛开大量的判断逻辑和类调用，它主要完成了三个主要任务：

- 安装Keystone软件包
- 管理Keystone.conf中的主要配置项
- 管理Keystone服务

2.1.1 keystone软件包管理

这里有一个重要参数$package_ensure，可以指定软件包的版本，或者将其标记为总是安装最新版本，本书将会在最佳实践部分再次提及它。

```
# keystone软件包
  package { 'keystone':
    ensure => $package_ensure,
    name   => $::keystone::params::package_name,
    tag    => ['openstack', 'keystone-package'],
  }
# keystone-client软件包
  if $client_package_ensure == 'present' {
    include '::keystone::client'
  } else {
    class { '::keystone::client':
      ensure => $client_package_ensure,
    }
  }
```

2.1.2 keystone.conf核心参数管理

class keystone管理了大量的配置项，比如cache, token, db, endpoint等相关参数，这里不一一列举。

那么对于这些选项是如何管理的呢？这里我们要提到 `keystone_config` 。

`keystone_config` 是一个自定义的resource type，其代码路径是：

- lib/puppet/type/keystone_config.rb 定义了keystone_config
- lib/puppet/provider/keystone_config/ini_setting.rb 实现了keystone_config

在这里我们关注如何使用 `keystone_config` 。

keystone_config有几种使用场景:

对指定参数赋值：

```
keystone_config { 'section_name/option_name': value => option_va
```

对指定参数赋值，并设置为加密：

```
keystone_config { 'section_name/option_name': value => option_va
```

我们知道puppet agent的所有输出默认都会被syslog打到系统日志/var/log/messages中，那么有心人只要用grep就能从中搜到许多敏感信息，例如：admin_token, user_password, keystone_db_password等等。只要设置了secret为true后，那么就不会把该参数的相关日志打到系统日志中。

删除指定参数:

```
keystone_config { 'section_name/option_name': ensure => absent}
```

OK，讲解就到这里，下面看一段实际的代码。

```
keystone_config {
  'DEFAULT/admin_token':       value => $admin_token, secret => tr
  'DEFAULT/public_bind_host': value => $public_bind_host;
  'DEFAULT/admin_bind_host':  value => $admin_bind_host;
  'DEFAULT/public_port':      value => $public_port;
  'DEFAULT/admin_port':       value => $admin_port;
}
```

与之对应的keystone.conf配置文件[DEFAULT]下的admin_token等配置项被Puppet修改为指定值。

2.1.3 keystone服务管理

puppet支持keystone以单进程模式运行或者跑在Apache上，请注意，如果需要将keystone运行在Apache上，那么需要添加keystone::wsgi::apache，代码如下：

```
class { 'keystone':
   ...
   service_name => 'httpd',
   ...
}
class { 'keystone::wsgi::apache':
   ...
}
```

我们来看一下管理keystone服务的逻辑：

```
if $service_name == $::keystone::params::service_name {
   $service_name_real = $::keystone::params::service_name
    ...
     #这里调用了keystone::service类，用于管理keystone服务的具体配置
     class { '::keystone::service':
       ensure         => $service_ensure,
       service_name   => $service_name,
       enable         => $enabled,
       hasstatus      => true,
       hasrestart     => true,
       validate       => true,
       admin_endpoint => $v_auth_url,
       admin_token    => $admin_token,
       insecure       => $validate_insecure,
       cacert         => $validate_cacert,
     }
   } else {
     class { '::keystone::service':
       ensure       => $service_ensure,
       service_name => $service_name,
       enable       => $enabled,
       hasstatus    => true,
       hasrestart   => true,
       validate     => false,
     }
```

```
    }
    warning('Keystone under Eventlet has been deprecated during the
  } elsif $service_name == 'httpd' {
    # 在这里，我们可以看到当$service_name为httpd时，将keystone service的
    include ::apache::params
    class { '::keystone::service':
      ensure       => 'stopped',
      service_name => $::keystone::params::service_name,
      enable       => false,
      validate     => false,
    }
    $service_name_real = $::apache::params::service_name
    # leave this here because Ubuntu packages will start Keystone a
    # before apache can run
    Service['keystone'] -> Service[$service_name_real]
  } else {
      fail('Invalid service_name. Either keystone/openstack-keystor
  }
```

2.2 class keystone::service

在 `class keystone` 中就遇到了keystone::service，从类的名称可以得知，该类用于管理Keystone服务。其中有两段代码需要注意：

第一段是管理keystone服务：

```
  service { 'keystone':
    ensure     => $ensure,
    name       => $service_name,
    enable     => $enable,
    hasstatus  => $hasstatus,
    hasrestart => $hasrestart,
    tag        => 'keystone-service',
  }
```

第二段代码比较有意思，类似于smoketest，简单调用keystone的user list接口来验证keystone服务是否正常运行：

```
  if $validate and $admin_token and $admin_endpoint {
    $cmd = "openstack --os-auth-url ${admin_endpoint} --os-token $
    $catch = 'name'
    exec { 'validate_keystone_connection':
      path        => '/usr/bin:/bin:/usr/sbin:/sbin',
      provider    => shell,
      command     => $cmd,
      subscribe   => Service['keystone'],
      refreshonly => true,
      tries       => $retries,
      try_sleep   => $delay,
      notify      => Anchor['keystone::service::end'],
    }
  }
```

2.3 class keystone::endpoint

顾名思义，`class keystone::endpoint` 用于创建和管理Keystone的service和endpoint。

以下是使用样例：

```
class { 'keystone::endpoint':
  public_url   => 'https://154.10.10.23:5000',
  internal_url => 'https://11.0.1.7:5000',
  admin_url    => 'https://10.0.1.7:35357',
}
```

那么它是如何实现的呢？继续往下看，它又调用了一个define。

```
keystone::resource::service_identity { 'keystone':
  configure_user       => false,
  configure_user_role => false,
  service_type         => 'identity',
  service_description => 'OpenStack Identity Service',
  public_url           => $public_url_real,
  admin_url            => $admin_url_real,
  internal_url         => $internal_url_real,
  region               => $region,
  user_domain          => $user_domain,
  project_domain       => $project_domain,
  default_domain       => $default_domain,
}
```

接下来，需要跳转到 `keystone::resource::service_identity` 的定义了。

2.3.1 define keystone::resource::service_identity

莫慌，接着来看keystone::resource::service_identity，终于到路的尽头了，来看看它是怎么实现的。

首先，它实现了管理keystone user：

```
if $configure_user {
    if $user_domain_real {
      # We have to use ensure_resource here and hope for the best,
      # no way to know if the $user_domain is the same domain passe
      # $default_domain parameter to class keystone.
      ensure_resource('keystone_domain', $user_domain_real, {
        'ensure'  => 'present',
        'enabled' => true,
      })
    }
    ensure_resource('keystone_user', $auth_name, {
      'ensure'                => 'present',
      'enabled'               => true,
      'password'              => $password,
      'email'                 => $email,
      'domain'                => $user_domain_real,
    })
    if ! $password {
      warning("No password had been set for ${auth_name} user.")
    }
  }
```

这里的关键是keystone_user资源，这又是一个自定义resource type，其源码路径为:

- lib/puppet/type/keystone_config.rb 定义
- lib/puppet/provider/keystone_config/ini_setting.rb 实现

通过keystone_user，Puppet完成了对user的管理（包括创建,修改,查询）。

同理，还有keystone_domain，目的是完成对domain的管理。剩下的代码同理，就不一一解读了。

3.小结

在这里，我们介绍了puppet-keystone的核心代码，当然该module还有许多重要的class我们并没有涉及，例如：keystone::deps，keystone::policy等等。这些就留给读者自己去阅读代码了。

4.动手练习

1. 配置token_flush的cron job，使得可以定期清理Keystone数据库的token表中token失效数据。
2. 将keystone服务运行在Apache上
3. 如何开启keystone的debug日志级别
4. 接第3问，在keystone和keystone::loging里都存在$verbose变量，这种代码冗余的原因是出于什么考虑？可以移除吗？

puppet-nova 模块介绍

本节作者：廖鹏辉

阅读级别：必读

阅读时间 **1.5h**

`puppet-nova` 模块用于配置和管理Nova服务，包括服务，软件包，配置文件，flavor，nova cells等等。其中 nova flavor, cell 等资源的管理是使用自定义的resource type来实现的。

先睹为快

Nova 服务内部有很多组件，其中最重要的组件是 nova-api 和 nova-compute，这两个服务的部署在 nova 模块中都有专门的类来完成，当然在部署之前环境中需要有 keystone 来为 nova 提供认证服务。

以部署 nova-api 为例：

```
class { 'nova':
  rabbit_host          => 'localhost',
  rabbit_password      => 'password',
  rabbit_userid        => 'user',
  database_connection => 'mysql://nova:nova_pass@localhost/nova?cha
}
class nova::keystone::auth {
  password => 'password',
}
class nova::api {
  admin_password => 'password',
}
```

即可完成 nova-api 的基本部署。其中 `nova` 这个类主要负责所有 nova 服务通用配置项的配置，`nova::keystone::auth` 用于创建 keystone 用户，服务，endpoint，以及角色和用户的关联，`nova::api` 用于部署 nova-api 服务，管理相关的配置文件，并管理 nova-api 服务。

核心代码讲解

class nova

Nova 是一个有多个内部组件的 OpenStack 服务，这些服务可以分开部署在不同的节点中，服务之间使用消息队列进行通信，有些组件会使用到数据库，还可能和 keystone 服务进行交互。nova 虽然服务众多，但是配置文件只有一份，这个配置文件中所有服务通用的配置项，也有某个服务特有的配置项，对于通用的这些配置项，主要使用 `nova` 这类来进行管理，这个类主要管理了这些选项：

- 数据库相关的配置
- 消息队列相关的配置
- 日志相关的配置
- SSL 相关的配置

这个类主要使用 `nova_config` 来对这些配置进行管理，它的使用也非常简单，只用传递相关的参数就可以了。

class nova::keystone::auth

这个类的主要功能是添加 keystone 用户，以及用户和角色的关联，它通过调用 keystone 模块的 `keystone::resource::service_identity` 这个 define 资源来完成所有 keystone 中资源的创建。

```
keystone::resource::service_identity { "nova service, user ${auth
  configure_user      => $configure_user,
  configure_user_role => $configure_user_role,
  configure_endpoint  => $configure_endpoint,
  service_type        => 'compute',
  service_description => $service_description,
  service_name        => $real_service_name,
  region              => $region,
  auth_name           => $auth_name,
  password            => $password,
  email               => $email,
  tenant              => $tenant,
  public_url          => $public_url_real,
  admin_url           => $admin_url_real,
  internal_url        => $internal_url_real,
}
```

class nova::api

nova::api 这个类用来配置和管理 nova-api 服务以及相应的配置，其中比较重要的是用于 keystone 认证的相关配置。

首先，代码中会使用 `nova::generic_service` 来完成 nova-api 这个软件包的安装和服务的管理，`nova::generic_service` 这个资源主要的作用是管理 nova 中各个组件的软件包安装和服务的启动：`

```
nova::generic_service { 'api':
    enabled        => $service_enabled,
    manage_service => $manage_service,
    ensure_package => $ensure_package,
    package_name   => $::nova::params::api_package_name,
    service_name   => $::nova::params::api_service_name,
    subscribe      => Class['cinder::client'],
}
```

然后通过 `nova_config` 和 `nova_paste_api_ini` 这个两个自定义资源来对 `/etc/nova/nova.conf` 和 `/etc/nova/api-paste.ini` 进行一系列的配置，并通过 `nova::db` 来进行数据库相关的配置。

class nova::conductor

nova::conductor 这个类比较简单，主要使用 `nova::generic_service` 来完成 conductor 服务和软件包的管理，并配置了 workers 参数。

class nova::compute

nova::compute 这个类用来配置 nova-compute 服务，nova-compute 服务一般部署在计算节点，用于完成虚拟机的创建。

nova::compute 中主要完成了：

- nova-compute 相关配置的管理，如 VNC，网络相关的配置
- 管理 nova-compute 的软件包和服务

这些服务和配置的管理都是通过 `nova_config` 和 `nova::generic_service` 两个资源来完成的。

class nova::migration::libvirt

nova 可以控制 Libvirt 来完成虚拟机的迁移，配置虚拟机迁移除了需要配置 nova 的配置之外，还需要配置 Libvirt 相关的配置，因此在 nova 模块中专门有一个 `nova::migration::libvirt` 的类来进行 libvirt 相关的配置，这个类中，使用了 `augeas` 和 `file_line` 资源来配置 `/etc/libvirt/libvirtd.conf` ：

```
    augeas { 'libvirt-conf-uuid':
      context => '/files/etc/libvirt/libvirtd.conf',
      changes => [
        "set host_uuid ${host_uuid}",
      ],
      notify  => Service['libvirt'],
      require => Package['libvirt'],
    }
  }
```

`augeas` 能够将配置文件当做树形的结构来进行处理，详细的使用说明可以参考这里。同时，也使用了 `file_line` 来进行 libvirtd.conf 的配置：

```
      file_line { '/etc/libvirt/libvirtd.conf listen_tls':
        path  => '/etc/libvirt/libvirtd.conf',
        line  => "listen_tls = ${listen_tls}",
        match => 'listen_tls =',
        tag   => 'libvirt-file_line',
      }
```

小结

puppet-nova 模块中的内容众多，按照 nova 中的各个服务和功能进行了拆分，每个服务都有对应的 puppet 类进行管理，模块中还包含了 neutron, nova cell 等资源的管理，感兴趣的读者可以研究模块中其余的代码。

动手练习

1. nova 中的各个服务是通过哪个统一的自定义资源进行管理的？阅读这个 define 资源的代码，查看它的实现方式。
2. 部署 nova-api, nova-scheduler, nova-conductor 服务
3. 如何设置 nova-compute 服务的宿主机内存分配比，这些资源分配比例的设定是在哪个类中进行管理的？
4. 如何将 nova-compute

- puppet-nova 模块介绍

puppet-neutron 模块介绍

本节作者：廖鹏辉

建议阅读时间 **1.5h**

neutron 组件是 OpenStack 各组件中最为复杂的组件，puppet-neutron 模块提供了 neutron 各个组件的部署和管理，包括 neutron plugins 的管理和 neutron agents 的管理。

先睹为快

Neutron 是一个分布式的服务，它由 neutron-server 和不同功能的 agent 组成。neutron-server 用于处理 API 请求，agent 用来完成各种网络功能。

以部署 neutron-server 为例：

```
class { '::neutron::keystone::auth':
  public_url   => "http:/localhost:9696",
  internal_url => "http://localhost:9696",
  admin_url    => "http://localhost:9696",
  password     => 'a_big_secret',
}
class { '::neutron':
  rabbit_user           => 'neutron',
  rabbit_password       => 'an_even_bigger_secret',
  rabbit_host           => localhost,
  rabbit_port           => 5672,
  core_plugin           => 'ml2',
}
class { '::neutron::client': }
class { '::neutron::server':
  database_connection => 'mysql+pymysql://neutron:neutron@127.0.0
  password            => 'a_big_secret',
}
```

这里使用了 `neutron` ， `neutron::client` ， `neutron::server` ， `neutron::keystone::auth` 四个类，分别用于完成服务的基础配置，客户端的安装，neutron-server 服务的配置和管理，以及 keystone 的认证。

核心代码讲解

class neutron

这个类主要完成一些通用的 neutron 配置，主要包括有：

- 消息队列相关的配置
- 日志相关的配置
- SSL 相关的配置

这个类主要使用 `neutron_config` 来对这些配置进行管理，同时还使用了 `oslo` 模块来完成消息队列相关的配置管理。

class neutron::keystone::auth

这个类的主要功能是添加 keystone 用户，以及用户和角色的关联，它通过调用 keystone 模块的 `keystone::resource::service_identity` 这个 define 资源来完成所有 keystone 中资源的创建。

```
keystone::resource::service_identity { $auth_name:
  configure_user      => $configure_user,
  configure_user_role => $configure_user_role,
  configure_endpoint  => $configure_endpoint,
  service_type        => $service_type,
  service_description => $service_description,
  service_name        => $real_service_name,
  region              => $region,
  password            => $password,
  email               => $email,
  tenant              => $tenant,
  public_url          => $public_url,
  admin_url           => $admin_url,
  internal_url        => $internal_url,
}
```

class neutron::server

nova::server 用来管理 neutron-server 服务，这个服务是 neutron 的核心服务，用于处理 API 请求。代码中主要使用 `neutron_config` 来完成 keystone 用户认证相关的配置，数据库连接相关的配置，以及一些 agent 的基础配置。

这个类的代码中多次了 `ensure_resource` 函数来创建资源，这样做的好处是 `ensure_resource` 在创建资源前会检查是否有重复的资源定义，如果有重复的资源定义那么就不再重复创建资源，可以避免资源的重复定义，我们来看一些这个函数是如何被使用的：

```
if $ensure_vpnaas_package {
  ensure_resource( 'package', 'neutron-vpnaas-agent', {
    'ensure' => $package_ensure,
    'name'   => $::neutron::params::vpnaas_agent_package,
    'tag'    => ['openstack', 'neutron-package'],
  })
}
```

这里使用了 package 资源来进行软件包的管理，如果相同的资源已经定义过了，那么 `ensure_resource` 函数将不再重复创建此资源。

class neutron::plugins::ml2

neutron::plugins::ml2 用于配置 ml2 plugin 相关的配置，包括 `/etc/neutron/plugin.ini` 软链接的创建，服务配置项的管理等等。关于 ml2 plugin 的配置，在 neutron 中有专用的自定义资源 `neutron_plugin_ml2` 用来配置 ml2 的配置文件：

```
neutron_plugin_ml2 {
  'ml2/type_drivers':                        value => join(any2array
  'ml2/tenant_network_types':                value => join(any2array
  'ml2/mechanism_drivers':                   value => join(any2array
  'ml2/path_mtu':                            value => $path_mtu;
  'ml2/extension_drivers':                   value => join(any2array
  'securitygroup/enable_security_group':     value => $enable_secur
  'securitygroup/firewall_driver':           value => $firewall_driv
}
```

并且，还控制了 ml2-plugin 软件包的安装顺序，在安装完软件包之后才应该配置其相关配置文件：

```
  if $::neutron::params::ml2_server_package {
    package { 'neutron-plugin-ml2':
      ensure => $package_ensure,
      name   => $::neutron::params::ml2_server_package,
      tag    => 'openstack',
    }
    Package['neutron-plugin-ml2'] -> File['/etc/neutron/plugin.ini
    Package['neutron-plugin-ml2'] -> File['/etc/default/neutron-se
    Package['neutron-plugin-ml2'] -> Neutron_plugin_sriov<||>
  } else {
    Package['neutron'] -> File['/etc/neutron/plugin.ini']
    Package['neutron'] -> File['/etc/default/neutron-server']
    Package['neutron'] -> Neutron_plugin_sriov<||>
  }
```

class neutron::agents::ml2::ovs

openvswitch-agent 是使用 neutron 使最常用的 agent，它通常被部署在网络节点和计算节点，用来完成 ovs bridge 的管理，ovs-agent 由 neutron::agents::ml2::ovs 这个类来进行管理，neutron 模块中为 ovs-agent 的配置提供了专门的自定义资源 `neutron_agent_ovs` 用于管理其配置文件，这个类中，主要使用了 `neutron_agnet_ovs` 来完成 ovs-agent 的配置，并管理了 ovs-agent 的软件包和服务。与这个类类似的，还有 `neutron::agents::ml2::linuxbridge` 用来管理 linuxbridge-agent 相关的配置和服务。

class neutron::agents::l3

l3-agent 通常部署在网络节点，提供网络间转发与路由的功能，`neutron::agents::l3` 这个类用于完成 L3-agent 的配置与管理。值得注意的是，它在代码中，使用了 `is_service_default` 这个函数：

```
  if ! is_service_default ($external_network_bridge) {
    warning('parameter external_network_bridge is deprecated')
  }

  if ! is_service_default ($router_id) {
    warning('parameter router_id is deprecated and will be removed
  }
```

这个函数是在 puppet-openstacklib 中，定义的，它的作用是判断一个变量的值是否等于 $::os_service_default 这个 facter 的值，即这个变量是否为默认值。这里对一些废弃参数的值进行了判断，如果用户修改了这些废弃参数的值，那么将会收到 warning 警告信息，告诉用户这个参数已经被废弃了。我们可以看到 `is_service_default` 这个函数的定义如下：

```
module Puppet::Parser::Functions
  newfunction(:is_service_default, :type => :rvalue, :doc => <<-EOS
Returns true if the variable passed to this function is '<SERVICE D
  EOS
  ) do |arguments|
    raise(Puppet::ParseError, "is_service_default(): Wrong number o
          "given (#{arguments.size} for 1)") if arguments.size != 1

    value = arguments[0]

    unless value.is_a?(String)
      return false
    end

    return (value == '<SERVICE DEFAULT>')
  end
end
```

这个函数首先对传递的参数个数进行了检查，然后比较了参数类型是否为字符串，最后将参数与 `'<SERVICE DEFAULT>'` 进行比较，并返回布尔值。

小结

puppet-neutron 模块管理了 neutron 的 neutron-server 服务，各种 plugin以及不同的 agent 服务，同时模块总还有管理其他服务如 lbaas, vpnaas 等服务的专用类，读者可以自行去探究其代码。

动手练习

1. 部署 neutron lbaas 服务，查看 neutron 模块中有哪些类是用来管理这个服务相关组件的
2. 使用 `neutron_port` 和 `neutron_router` 自定义资源来创建 neutron port 和 router

puppet-glance 模块

0. 项目简介

Glance是OpenStack Image Service项目，用于注册、管理和检索虚拟机镜像。Glance并不负责实际的镜像存储。它提供了对接简单文件系统，对象存储，块存储等多种存储后端的能力。除了磁盘镜像信息，它还能够存储描述镜像的元数据和状态信息。

1.先睹为快

不想看下面大段的代码解析，已经跃跃欲试了？

OK，我们开始吧！

创建puppet_glance.pp文件并输入:

```
class { 'glance::api':
  verbose             => true,
  keystone_tenant     => 'services',
  keystone_user       => 'glance',
  keystone_password   => '12345',
  database_connection => 'mysql://glance:12345@127.0.0.1/glance',
}

class { 'glance::registry':
  verbose             => true,
  keystone_tenant     => 'services',
  keystone_user       => 'glance',
  keystone_password   => '12345',
  database_connection => 'mysql://glance:12345@127.0.0.1/glance',
```

```
}

class { 'glance::backend::file': }

class { 'glance::db::mysql':
  password      => '12345',
  allowed_hosts => '%',
}

class { 'glance::keystone::auth':
  password         => '12345'
  email            => 'glance@example.com',
  public_address   => '127.0.0.1',
  admin_address    => '127.0.0.1',
  internal_address => '172.17.1.3',
  region           => 'example-west-1',
}

rabbitmq_user { 'glance':
  admin    => true,
  password => 'an_even_bigger_secret',
  provider => 'rabbitmqctl',
  require  => Class['::rabbitmq'],
}

rabbitmq_user_permissions { 'glance@/':
  configure_permission => '.*',
  write_permission     => '.*',
  read_permission      => '.*',
  provider             => 'rabbitmqctl',
  require              => Class['::rabbitmq'],
}
```

在终端执行以下命令:

```
$ puppet apply -v puppet_glance.pp
```

2.核心代码讲解

2.1 class glance

`class glance` 用于管理Glance软件包和Openstackclient软件包:

```
include ::glance::params

if ( $glance::params::api_package_name == $glance::params::regist
  package { $::glance::params::api_package_name :
    ensure => $package_ensure,
    name   => $::glance::params::api_package_name,
    tag    => ['openstack', 'glance-package'],
  }
  include '::openstacklib::openstackclient'
}
```

2.2 class glance::api

`glance::api` 类用于管理以下配置：

1. **policy\db\logging\cache**

```
include ::glance::policy
include ::glance::api::db
include ::glance::api::logging
include ::glance::cache::logging
```

2. **/etc/glance/glance-api.conf**

```
# basic service config
glance_api_config {
  'DEFAULT/bind_host':                value => $bind_host;
  'DEFAULT/bind_port':                value => $bind_port;
  'DEFAULT/backlog':                  value => $backlog;
  'DEFAULT/show_image_direct_url':    value => $show_image_direct
   ...
  'DEFAULT/image_cache_dir':          value => $image_cache_dir;
  'DEFAULT/auth_region':              value => $auth_region;
  'glance_store/os_region_name':      value => $os_region_name;
}
```

3.管理**/etc/glance/glance-cache.conf**

在Glance-api中,启用Glance的缓存功能可以加速镜像的二次下载速度(注：在使用Ceph作为Glance, Cinder, Nova的后端时，此功能无效)

```
glance_cache_config {
  'DEFAULT/image_cache_stall_time': value => $image_cache_stall_t
  'DEFAULT/image_cache_max_size':   value => $image_cache_max_siz
  'glance_store/os_region_name':    value => $os_region_name;
}
```

4.**glance-api**服务的管理

```
service { 'glance-api':
  ensure     => $service_ensure,
  name       => $::glance::params::api_service_name,
  enable     => $enabled,
  hasstatus  => true,
  hasrestart => true,
  tag        => 'glance-service',
}
```

5.验证**glance-api**服务部署是否成功 通过调用 `glance image-list` 命令来验证glance-api的返回值是否符合预期。

```
  if $validate {
    $defaults = {
      'glance-api' => {
        'command'  => "glance --os-auth-url ${auth_uri} --os-tenant
      }
    }
    $validation_options_hash = merge ($defaults, $validation_option
    create_resources('openstacklib::service_validation', $validatio
  }
```

2.3 Class glance::registry

`glance::registry` 用于安装和配置 `glance-registry` 服务，其代码结构与 `glance::api` 类似，在此不做赘述。

Class glance::notify::rabbitmq

在glance-api和glance-registry中启用notifications功能可以在创建镜像，更新镜像源数据等事件发生时发送通知到rabbitmq给其他服务使用。

调用**puppet-oslo**来配置**glance-api.conf**和**glance-registry**

```
oslo::messaging::rabbit { ['glance_api_config', 'glance_registry_
  rabbit_password              => $rabbit_password,
  rabbit_userid                => $rabbit_userid,
  rabbit_host                  => $rabbit_host,
  rabbit_port                  => $rabbit_port,
  rabbit_hosts                 => $rabbit_hosts,
  rabbit_virtual_host          => $rabbit_virtual_host,
  rabbit_ha_queues             => $rabbit_ha_queues,
  heartbeat_timeout_threshold => $rabbit_heartbeat_timeout_thresh
  heartbeat_rate               => $rabbit_heartbeat_rate,
  rabbit_use_ssl               => $rabbit_use_ssl,
  kombu_ssl_ca_certs           => $kombu_ssl_ca_certs,
  kombu_ssl_certfile           => $kombu_ssl_certfile,
  kombu_ssl_keyfile            => $kombu_ssl_keyfile,
  kombu_ssl_version            => $kombu_ssl_version,
  kombu_reconnect_delay        => $kombu_reconnect_delay,
  amqp_durable_queues          => $amqp_durable_queues,
  kombu_compression            => $kombu_compression,
}


oslo::messaging::notifications { ['glance_api_config', 'glance_re
  driver => $notification_driver,
  topics => $rabbit_notification_topic,
}
```

2.4 `Class glance::backend::rbd`

Glance支持多种存储后端，比如cinder,swift,file,ceph,s3，本节将介绍如何使用 `glance::backend::rbd` 配置Ceph作为Glance后端存储：

```
#修改glance_store下的配置项
glance_api_config {
  'glance_store/rbd_store_ceph_conf':    value => $rbd_store_ceph
  'glance_store/rbd_store_user':         value => $rbd_store_user
  'glance_store/rbd_store_pool':         value => $rbd_store_pool
  'glance_store/rbd_store_chunk_size':   value => $rbd_store_chun
  'glance_store/rados_connect_timeout':  value => $rados_connect_
}

if !$multi_store {
  glance_api_config { 'glance_store/default_store': value => 'rbd
  if $glare_enabled {
    glance_glare_config { 'glance_store/default_store': value =>
  }
}

#安装python-ceph软件包
package { 'python-ceph':
  ensure => $package_ensure,
  name   => $::glance::params::pyceph_package_name,
}
```

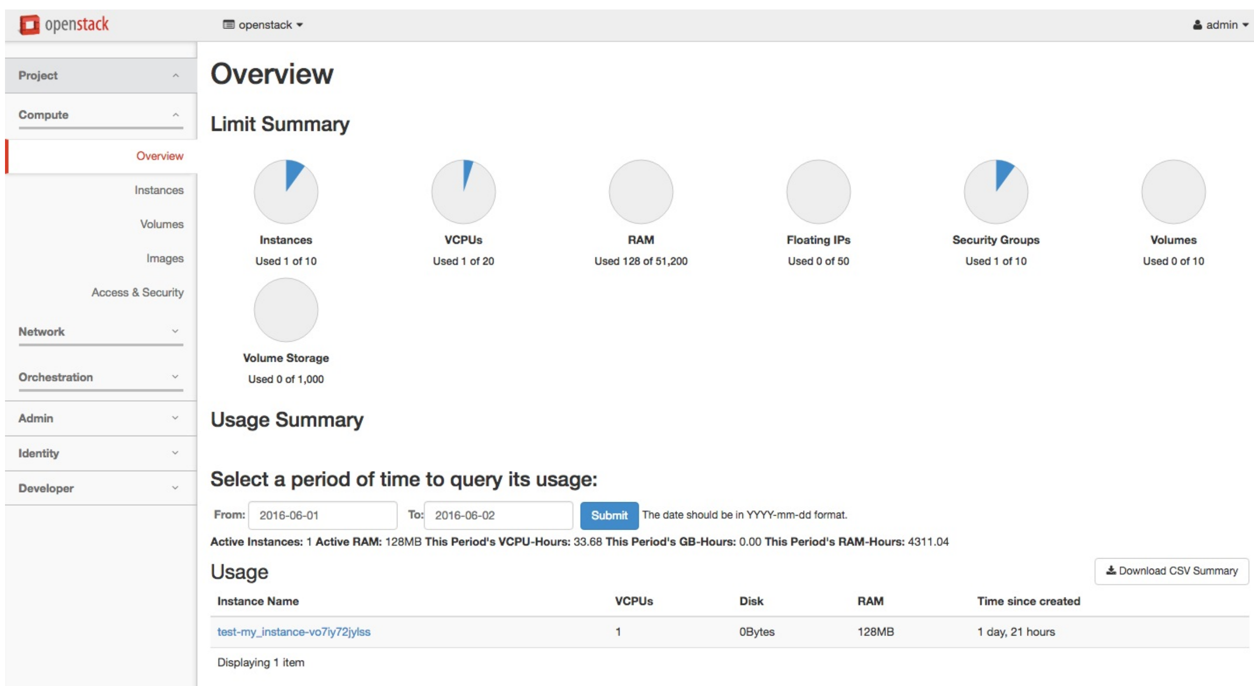
3.动手练习

1. 配置Glance使用Swift作为存储后端
2. 设置token的缓存时间为5min

puppet-horizon

0.基础知识

Horizon是OpenStack Dashbaord项目，为用户提供了Web图形化的管理界面来完成一些常见的虚拟资源操作，例如创建虚拟机实例，管理网络，设置访问权限等等。下图给出了Horizon的预览页面的样例。



除了四大核心项目以外，Horizon还支持以下项目：

- swift
- cinder
- heat
- ceilometer

- trove
- sahara

> 注：要正常运行horizon服务，至少需安装Nova,Keystone,Glance,Neutron服务

`puppet-horizon` 模块用于配置和管理horzion服务，包括Horzion软件包，配置文件和服务的管理，并且 `puppet-horizon` 支持将Horizon将运行在Python内置Web服务器或Apache服务器上。

1.先睹为快

不想看下面大段的代码解析，已经跃跃欲试了？

OK，我们开始吧！

打开虚拟机终端并输入以下命令：

```
$ puppet apply -e 'class {'horizon': secret_key => 'big'}'
```

等待命令执行完成，Puppet完成了Horizon部署，并将其运行在Apache上。

2.核心代码讲解

2.1 class horizon

`class horizon` 管理了以下三个任务:

- Horizon软件包的安装:

```
  package { 'horizon':
    ensure => $package_ensure,
    name   => $::horizon::params::package_name,
    tag    => ['openstack', 'horizon-package'],
  }
```

- Horizon配置文件的管理:

```
concat { $::horizon::params::config_file:
  mode    => '0644',
  require => Package['horizon'],
}

concat::fragment { 'local_settings.py':
  target  => $::horizon::params::config_file,
  content => template($local_settings_template),
  order   => '50',
}
```

这里说明一下concat管理配置文件的方式，在其他模块中也出现过这种管理配置文件的方式，而它曾经流行过一段时间。

了解过template的同学都知道，这是puppet管理配置文件的内置方式，这种方式的优缺点非常明显，其缺点就是每次配置文件发生新的变动，那么模板也得保持同步的更新。

因此，有人提出来一种新方法，将模板文件拆为分片(fragment)，把保持不变的配置项放到分片1中，把频繁更新的配置项放到分片2中，然后最后再拼接起来(concat)。这种方法简化了模板维护的成本，使得配置文件的管理变得更灵活，但本质上仍是模板。

- 管理Horizon服务的运行环境:

```
  if $configure_apache {
    class { '::horizon::wsgi::apache':
      bind_address   => $bind_address,
      servername     => $servername,
      server_aliases => $final_server_aliases,
      listen_ssl     => $listen_ssl,
      ssl_redirect   => $ssl_redirect,
      horizon_cert   => $horizon_cert,
      horizon_key    => $horizon_key,
      horizon_ca     => $horizon_ca,
      extra_params   => $vhost_extra_params,
      redirect_type  => $redirect_type,
      root_url       => $root_url
    }
  }
```

这个类的代码通俗易懂，值得一提的是以下4个参数，若配合不慎可能会导致服务运行异常：

- keystone_url
- available_regions
- cache_server_ip
- secret_key

还有一点是函数member的使用，类似于Python中的 `in` ，用于判断指定变量是否存在于指定的数组中，第一个参数是数组变量，第二个参数是成员变量：

```
 if ! (member($tuskar_ui_deployment_mode_allowed_values, $tuskar_ui
    fail("'${$tuskar_ui_deployment_mode}' is not correct value for
  }
```

2.1 `class horizon::wsgi::apache`

`horizon::wsgi::apache` 用于配置将horizon运行在apache上。

这里有两点值得注意，第一点是merge函数：

```
if $bind_address {
  $default_vhost_conf = merge($default_vhost_conf_no_ip, { ip =>
} else {
  $default_vhost_conf = $default_vhost_conf_no_ip
}
```

第二点是函数ensure_resource:

```
ensure_resource('apache::vhost', $vhost_conf_name, merge ($defaul
  redirectmatch_regexp => $redirect_match,
  redirectmatch_dest   => $redirect_url,
}))
```

其等价于:

```
$merged_hash_list = merge ($default_vhost_conf, $extra_params, {
   redirectmatch_regexp => $redirect_match,
   redirectmatch_dest   => $redirect_url,
 })
apache::vhost {"$vhost_conf_name":
   $merged_hash_list
 }))
```

使用 `ensure_resource` 的目的是为了使代码更简洁。

3.小结

本节介绍了如何使用 `puppet-horizon` 模块部署Horizon服务，同时也介绍了concat, merge, ensure_resource等define和function，合理使用有助于提高代码的简洁和优雅。

4.动手练习

1. 开启Horizon SSL端口

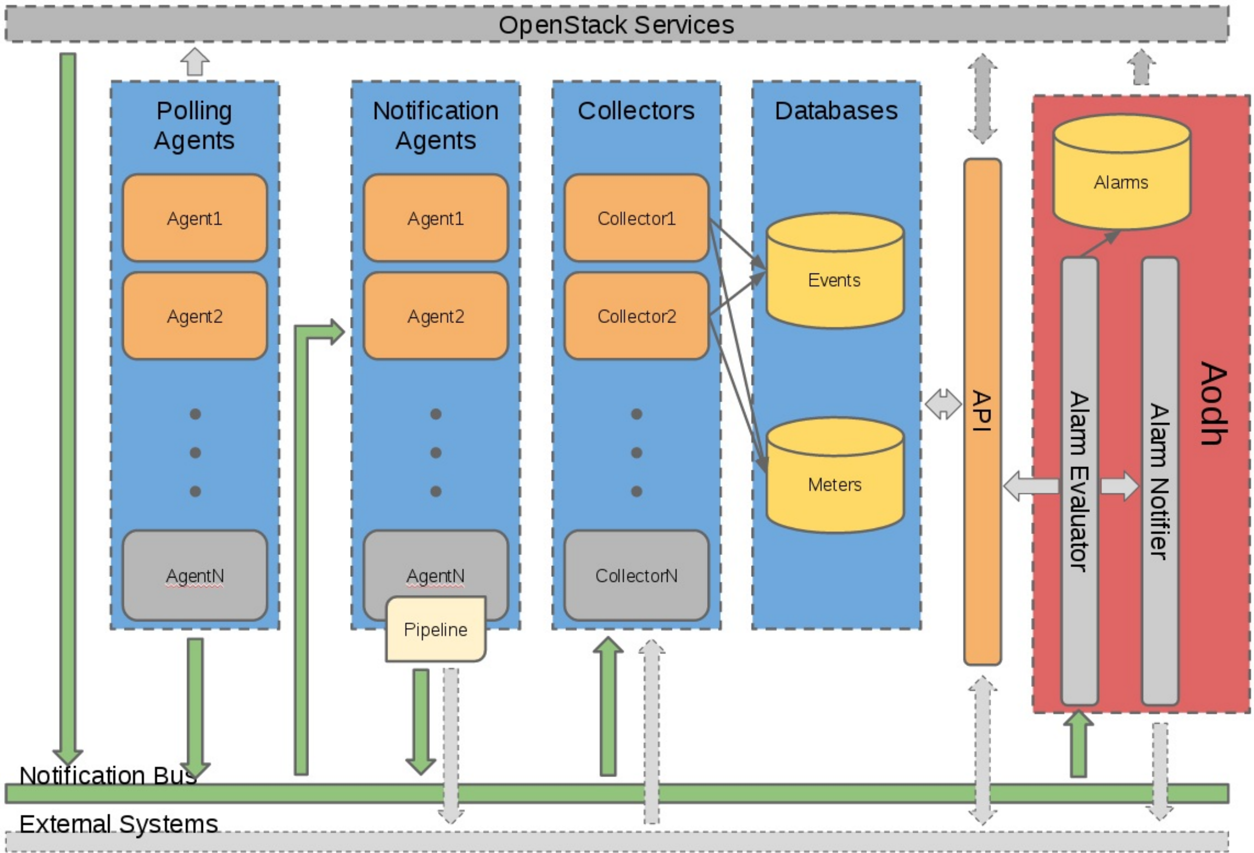
2. 确保Horizon服务只监听在内网IP上
3. 如何调整mod_wsgi的参数来设置Horizon运行在三种不同的MPM模式：prefork, worker, winnt

puppet-ceilometer

本节作者：韩亮亮

建议阅读时间 **1h**

ceilometer是openstack的数据收集模块，它把收集OpenStack内部发生的大部分事件，为计费和监控以及其它服务提供数据支撑。ceilometer的架构如下：



ceilometer服务

名称	说明
openstack-ceilometer-api	用于调用和查看collector收集的数据
openstack-ceilometer-collector	用于收集和记录polling和notification传过来的事件和计量数据
openstack-ceilometer-notification	用于监听消息队列，把感兴趣的监听消息变成Events和Samples，并发送到pipeline。
openstack-ceilometer-polling	用于获取openstack组件的信息，并生成监控数据，有三种启动类型：compute、central、ipmi。

ceilometer数据

名称	说明
Sample	某一个时间点获取到的监控数据
Event	就是一个事件、一个动作，如创建虚拟机、创建硬盘等。

ceilometer收集数据方式

名称	说明
Bus listener agent	通过监听消息队列来获取信息，官方首选。
Polling agents	通过调用API来收集信息。

先睹为快

由于ceilometer依赖很多服务，所以最好先部署一个openstack，我们可以使用下一站章节的puppet-openstack-integration或devstack部署一套简易版openstack。

部署ceilometer： 在examples/site.pp里添加下面的代码,因为默认的site.pp里没有创建endpoint,role。

```
class { 'ceilometer::keystone::auth':
  password      => 'tralalayouyou'        #这个参数是puppet-openst
}
```

然后执行以下命令

```
# puppet apply examples/site.pp
```

等一会ceilometer就安装完成了。 验证：

```
# source openrc
# ceilometer event-list
```

核心代码讲解

class ceilometer

class ceilometer中包括ceilometer组、用户的创建、软件包的安装，AMQP的选择及配置。

```
package { 'ceilometer-common':
  ensure => $package_ensure,
  name   => $::ceilometer::params::common_package_name,
  tag    => ['openstack', 'ceilometer-package'],
}
```

puppet-ceilometer中对rpc的选择主要提供了两种：RabbitMQ和amqp，所提供的参数如下:

```
  if $rpc_backend in [$::os_service_default, 'ceilometer.openstack.
    oslo::messaging::rabbit {'ceilometer_config':
      rabbit_host                 => $rabbit_host,
      rabbit_port                 => $rabbit_port,
      rabbit_hosts                => $rabbit_hosts,
      rabbit_userid               => $rabbit_userid,
      rabbit_password             => $rabbit_password,
      rabbit_virtual_host         => $rabbit_virtual_host,
      rabbit_ha_queues            => $rabbit_ha_queues,
......
    }
  } elsif $rpc_backend == 'amqp' {
    oslo::messaging::amqp { 'ceilometer_config':
      server_request_prefix  => $amqp_server_request_prefix,
      broadcast_prefix       => $amqp_broadcast_prefix,
      group_request_prefix   => $amqp_group_request_prefix,
      container_name         => $amqp_container_name,
      idle_timeout           => $amqp_idle_timeout,
      trace                  => $amqp_trace,
......
    }
  } else {
    nova_config { 'DEFAULT/rpc_backend': value => $rpc_backend }
  }
```

ceilometer通过调用oslo的oslo::messaging::notifications和oslo::cache两个define对oslo_messaging_notifications和cache两个section进行配置。

class ceilometer::api

在class ceilometer::api中先是定义了以下几个依赖关系：

```
Ceilometer_config<||> ~> Service[$service_name]
Class['ceilometer::policy'] ~> Service[$service_name]

Package['ceilometer-api'] -> Service[$service_name]
Package['ceilometer-api'] -> Class['ceilometer::policy']
```

在上面的代码中我们可以看到有两种符号'->'和'~>'，这两者都是描述依赖，只不过不同的是'->'是在执行完前面的资源 之后执行后面的资源，而'~>'则是如果前面的资源有变动执行后面的资源。 同时，ceilometer api支持两种管理方式，独立启动服务和通过httpd管理，默认是独立启动，我们可以通过给class ceilometer::api传递service_name参数进行修改，代码如下：

```
if $service_name == $::ceilometer::params::api_service_name {
  service { 'ceilometer-api':
    ensure     => $service_ensure,
    name       => $::ceilometer::params::api_service_name,
    enable     => $enabled,
    hasstatus  => true,
    hasrestart => true,
    require    => Class['ceilometer::db'],
    tag        => 'ceilometer-service',
  }
} elsif $service_name == 'httpd' {
  include ::apache::params
  service { 'ceilometer-api':
    ensure => 'stopped',
    name   => $::ceilometer::params::api_service_name,
    enable => false,
    tag    => 'ceilometer-service',
  }
  Class['ceilometer::db'] -> Service[$service_name]

  # we need to make sure ceilometer-api/eventlet is stopped befor
  Service['ceilometer-api'] -> Service[$service_name]
} else {
  fail('Invalid service_name. Either ceilometer/openstack-ceilome
}
```

其余代码则是对参数进行配置，略过。

class ceilometer::collector

class ceilometer::collector用于安装ceilometer的collector服务，依然是装包、配置文件、启动服务。 不过，在中间我们发现了一段代码，不是我们原先熟悉的package的方式安装，而是用了ensure_resource，这两种方式 的不同，在于当我们使用package的方式安装软件包，定义重复时会报错，而ensure_resource不会，代码如下：

```
ensure_resource( 'package', [$::ceilometer::params::collector_pac
  { ensure => $package_ensure }
)
```

class ceilometer::db

class ceilometer::db应该和db目录下的几个文件放在一起看，ceilometer默认使用MySQL数据库，首先ceilometer::db::mysql调用::openstacklib::db::mysql创建ceilometer的数据库，代码如下:

```
::openstacklib::db::mysql { 'ceilometer':
  user          => $user,
  password_hash => mysql_password($password),
  dbname        => $dbname,
  host          => $host,
  charset       => $charset,
  collate       => $collate,
  allowed_hosts => $allowed_hosts,
}
```

然后触发dbsync.而class ceilometer::db则调用oslo::db配置ceilometer中db相关参数。

```
oslo::db { 'ceilometer_config':
  db_max_retries => $database_db_max_retries,
  connection     => $database_connection,
  idle_timeout   => $database_idle_timeout,
  min_pool_size  => $database_min_pool_size,
  max_retries    => $database_max_retries,
  retry_interval => $database_retry_interval,
  max_pool_size  => $database_max_pool_size,
  max_overflow   => $database_max_overflow,
}
```

class ceilometer::keystone::auth

ceilometer::keystone::auth模块是用来创建ceilometer的endpoint和role，其中有这么一段代码：

```
::keystone::resource::service_identity { $auth_name:
  configure_user      => $configure_user,
  configure_user_role => $configure_user_role,
  configure_endpoint  => $configure_endpoint,
  service_type        => $service_type,
  service_description => $service_description,
  service_name        => $service_name_real,
  region              => $region,
  password            => $password,
  email               => $email,
  tenant              => $tenant,
  roles               => ['admin', 'ResellerAdmin'],
  public_url          => $public_url_real,
  admin_url           => $admin_url_real,
  internal_url        => $internal_url_real,
}
```

我们可以看到这里调用keystone::resource::service_identity这个define的时候前面有::符号，这就要讲到 puppet中的一个概念:域。 puppet中的域分为4种：顶级域、节点域、父域和本地域。在所有的类、定义或节点之外的就是顶级域，如在site.pp中定义了一个$v的参数，那我们可以在任意位置之中使用$::v来调用它。 节点定义

中节点名称后面的一对大括号就是节点域，节点域中定义的变量只能在该节点内调用。 父域和本地域的关系在于继承，如果class A通过关键字inherits引用了class B，如下：

```
class A{
  $variable = 'v1'
  ...
}
class B inherits A {
  ...
}
```

那么我们可以在class B中通过$::A::variable的方式调用该变量.

返过来看我们这段代码， ::keystone::resource::service_identity 这个调用前面使用::是在顶级域中搜索 keystone模块，这么看是不是就清晰多了。

class ceilometer::agent::auth

class ceilometer::agent::auth用于配置ceilometer中的keystone配置，默认密码没有配置，所以在调用该 class的时候必需传该参数。在class里会把传进的参数传到ceilometer_config,在class ceilometer::config 里调用。

class ceilometer::agent::polling

class ceilometer::agent::polling用于安装ceilometer polling agent,当然主要的还是那三板斧， 安装软件包、配置参数、启动服务。除这之外我们可以看到根据central_namespace、compute_namespace、ipmi_namespace三个参数，进行了不同的配置，并且通过inline_template调用ruby把namespaces这个数组转换 为字符串。代码如下：

```
  $namespaces = [$central_namespace_name, $compute_namespace_name,
  $namespaces_real = inline_template('<%= @namespaces.find_all {|x
```

inline_template用于在代码里使用嵌入式ruby，它里面的所有参数都会被传递并执行,在<%=和%>分隔符之间的所有代码都以Ruby代码来执行。

class ceilometer::agent::notification

这个class用于配置ceilometer的notification agent，没有什么好讲的，三步：安装软件包，启动服务，配置参数。

小结

在puppet-ceilometer模块中还有一些其他的class,如：ceilometer::policy、ceilometer::client、 ceilometer::config等，就留给读者自己去阅读了

动手练习

1. 安装ceilometer，并且安装compute和central两个客户端
2. 配置ceilometer运行在httpd下
3. 使用amqp替换RabbitMQ

puppet-cinder模块介绍

本节作者：周维宇

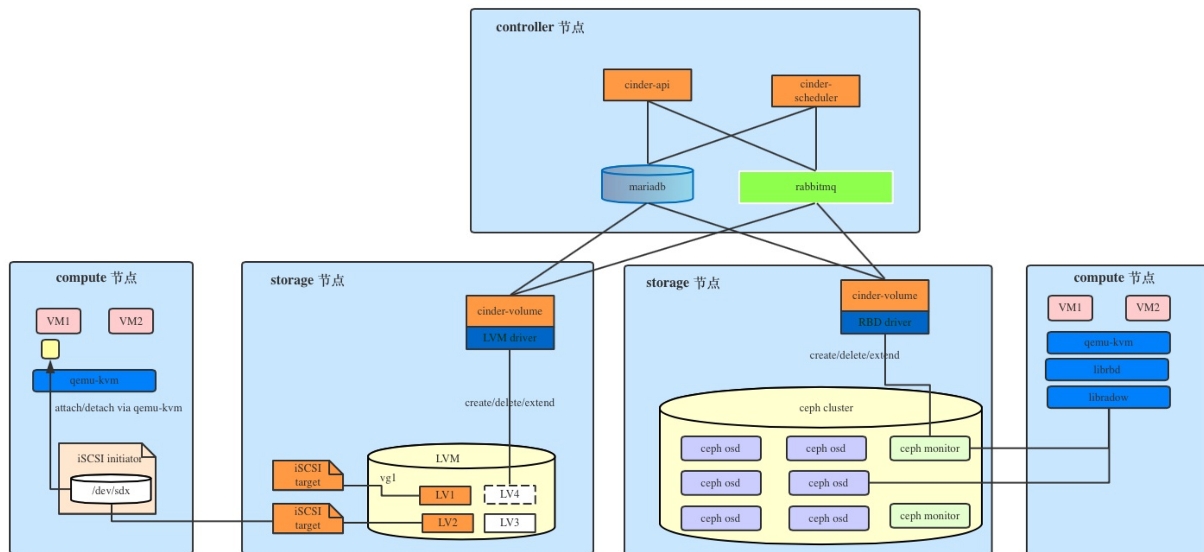
建议阅读时间 **2**小时

基础知识

cinder概述

- 它是一个资源管理系统，负责向虚拟机提供持久块存储资源(云硬盘)。
- 主要核心是对卷的管理，允许对卷，卷的类型，卷的快照进行处理
- 它把不同的后端存储进行封装，向外提供统一的API。
- 它不是新开发的块设备存储系统，而是使用插件的方式结合不同后端存储的驱动提供块存储服务。

cinder架构



上图环境是根据本章的实验环境绘制

主要组件介绍

- cinder-api：主要服务接口, 负责接受和处理外界的API请求，并将请求放入消息队列，交由其他组件执行。
- cinder-scheduer: 根据预定的策略选择合适的cinder-volume节点来处理用户的请求，如果用户请求中执行的具体的存储节点则不需要cinder-scheduler介入。
- cinder-volume: 该服务运行在存储节点，通过driver负责实际的卷管理工作。
- cinder-backup: 备份cinder卷到其他存储(swift,ceph等)。

实验环境说明

- cinder-api/mariadb/rabbitmq部署在控制节点,cinder-volume/ceph-monitor/ceph-osd部署在存储节点(你也可以把所有服务部署在同一节点)
- 部署了两个cinder－volume,一个使用ceph作为存储后端，一个使用lvm作为存储后端
- 本实验环境依赖前面章节部署的mariadb/keystone/ceph/rabbitmq

先睹为快

在讲解cinder模块之前让我们先使用puppet把我们的实验环境部署起来,请根据你的具体环境修改learn_cinder.pp

编写learn_cinder.pp

```
class { 'cinder':
  database_connection     => 'mysql://cinder:secret_block_password@
  rabbit_password         => 'secret_rpc_password_for_blocks',
  rabbit_host             => 'openstack-controller.example.com',
  verbose                 => true,
}

class { 'cinder::api':
  keystone_password       => $keystone_password,
  keystone_enabled        => $keystone_enabled,
  keystone_user           => $keystone_user,
  keystone_auth_host      => $keystone_auth_host,
  keystone_auth_port      => $keystone_auth_port,
  keystone_auth_protocol  => $keystone_auth_protocol,
  service_port            => $keystone_service_port,
  package_ensure          => $cinder_api_package_ensure,
  bind_host               => $cinder_bind_host,
  enabled                 => $cinder_api_enabled,
}

class { 'cinder::scheduler': }

class { 'cinder::volume': }

cinder::backend::rbd {'rbd-images':
  rbd_pool => 'images',
  rbd_user => 'images',
}

cinder_type {'ceph':
  ensure     => present,
  properties => ['volume_backend_name=rbd-images'],
}

class { 'cinder::backends':
  enabled_backends => ['iscsi1', 'iscsi2', 'rbd-images']
}
```

在终端执行以下命令:

```
puppet apply -v learn_cinder.pp
```

ok，恭喜你，已经有了一个使用ceph作为后端的cinder服务，敢紧来试试吧

```
source openrc
openstack volume create test_cinder --size 1 --type ceph
```

你已经创建了一个1G大小的cinder卷

核心代码讲解

Class cinder

class cinder非常简单主要做了两件核心工作

- 安装cinder基础包
- 配置cinder.conf中的核心参数

软件包管理

这里有一个非常有用的参数是$package_ensure，我们可以指定软件包的版本，或者将其标记为总是安装最新版本，我们将会在最佳实践部分去介绍它。

```
  package { 'cinder':
    ensure  => $package_ensure,
    name    => $::cinder::params::package_name,
    tag     => ['openstack', 'cinder-package'],
    require => Anchor['cinder-start'],
  }
```

cinder核心参数管理

class cinder里管理了大量的配置参数，比如db,rpc,az设置等相关参数，这里不一一列举。 这里只一个代码片段为例来解释cinder_config的用法。和前面介绍的keystone_config类似cinder_config是一个自定义的resource type，其源码路径位于：

lib/puppet/type/cinder_config.rb 定义

lib/puppet/provider/cinder_config/ini_setting.rb 实现

在这里我们关注如何使用，在Advanced Puppet一书中我们将讲解如何编写custom resource type。 cinder_config有多种使用方法:

对指定参数赋值：

```
cinder_config { 'section_name/option_name': value => option_value}
```

对指定参数赋值，并设置为加密：

```
cinder_config { 'section_name/option_name': value => option_value，
```

我们知道puppet agent的所有输出默认都会被syslog打到系统日志/var/log/messages中，那么有心人只要用grep就能从中搜到许多敏感信息，例如：admin_token, user_password, keystone_db_password等等。只要设置了secret为true后，那么就不会把该参数的相关日志打到系统日志中。 OK，讲解就到这里，我们来看代码。

```
  cinder_config {
    'DEFAULT/enable_v1_api':        value => $enable_v1_api;
    'DEFAULT/enable_v2_api':        value => $enable_v2_api;
    'DEFAULT/enable_v3_api':        value => $enable_v3_api;
  }
```

Class cinder::api

class cinder::api 主要配置和管理cinder的api服务

管理服务

cinder可以作为一个服务启动，也可以启动在apache下

```
if $service_name == $::cinder::params::api_service {
  service { 'cinder-api':
    ensure    => $ensure,
    name      => $::cinder::params::api_service,
    enable    => $enabled,
    hasstatus => true,
    require   => Package['cinder'],
    tag       => 'cinder-service',
  }

} elsif $service_name == 'httpd' {
  include ::apache::params
  service { 'cinder-api':
    ensure => 'stopped',
    name   => $::cinder::params::api_service,
    enable => false,
    tag    => ['cinder-service'],
  }
```

服务检查

调用cinder list命令来确确认cinder服务是否ready

```
if $validate {
  $defaults = {
    'cinder-api' => {
      'command'  => "cinder --os-auth-url ${auth_uri} --os-tenant
    }
  }
  $validation_options_hash = merge ($defaults, $validation_optior
  create_resources('openstacklib::service_validation', $validatic
}
```

Class cinder::scheduler

这个class没什么好讲的，无非是装包，改配置，启服务三板斧

Class cinder::volume

同上

Class cinder::backup

同上

Define cinder::backend::*backend_name*

后端的定义由很多define组成，我们举例我们用到的cinder::backend::rbd,比较值得注意的是用define来实现后端定义，因为在cinder中可能有多个同一类型的后端,比如一个cinder配置两个ceph作为cinder存储后端，这时候用class实现显然是不合适的 主要也是调用cinder_config 来修改cinder.conf文件

```
cinder_config {
  "${name}/volume_backend_name":                value => $volume_ba
  "${name}/volume_driver":                      value => 'cinder.vo
  "${name}/rbd_ceph_conf":                      value => $rbd_ceph_
  "${name}/rbd_user":                           value => $rbd_user;
  "${name}/rbd_pool":                           value => $rbd_pool;
  "${name}/rbd_max_clone_depth":                value => $rbd_max_c
  "${name}/rbd_flatten_volume_from_snapshot": value => $rbd_flatt
  "${name}/rbd_secret_uuid":                    value => $rbd_secre
  "${name}/rados_connect_timeout":              value => $rados_con
  "${name}/rados_connection_interval":          value => $rados_con
  "${name}/rados_connection_retries":           value => $rados_con
  "${name}/rbd_store_chunk_size":               value => $rbd_store
}
```

Class cinder::backends

由于cinder支持多后端，这个类主要用来管理开启哪些存储后端

调用cinder_config来修改cinder.conf

```
class cinder::backends (
  $enabled_backends    = undef,
) {

  # Maybe this could be extented to dynamicly find the enabled name
  cinder_config {
    'DEFAULT/enabled_backends': value => join($enabled_backends, ',
  }
}
```

Define cinder::type

cinder开启多后端后，如何确定要将卷创建到哪个后端呢，这就要有type来决定.

```
define cinder::type (
  $set_key         = undef,
  $set_value       = undef,
  # DEPRECATED PARAMETERS
  $os_password     = undef,
  $os_tenant_name = undef,
  $os_username     = undef,
  $os_auth_url     = undef,
  $os_region_name = undef,
  ) {

  if $os_password or $os_region_name or $os_tenant_name or $os_user
    warning('Parameters $os_password/$os_region_name/$os_tenant_nar
    warning('Auth creds will be used from env or /root/openrc file
  }

  if ($set_value and $set_key) {
    if is_array($set_value) {
      $value = join($set_value, ',')
    } else {
      $value = $set_value
    }
    cinder_type { $name:
      ensure     => present,
      properties => ["${set_key}=${value}"],
    }
  } else {
    cinder_type { $name:
      ensure     => present,
    }
  }
}
```

这个关键的是cinder_type,其源码路径为

lib/puppet/type/cinder_type.rb

lib/puppet/provider/cinder_type/openstack.rb

小结

ok，核心代码的解析就到这里，后面的像cinder::quota,cinder::policy,cinder::logging等配置就不在一一解析,留给读者课后去学习.总之puppet-cinder除了多后端配置和其他模块略有不同之外,其余部分都十分相似，是一个比较容易学习的模块.

动手练习

1.在另外一个节点部署一个cinder-volume，并使用lvm作存储后端。

2.创建一个新的volume-type lvm,将该type的卷存储到lvm后端。

3.创建一个type为lvm的卷，并挂载到虚机。

puppet-tempest

本节作者：余兴超

阅读级别：必读

阅读时间: **40**分钟

Tempest是Openstack的集成测试框架，它的实现基于python的unittest2测试框架和nose测试框架。Tempest通过Openstack client发起API请求，并且对API响应结果进行验证。

先睹为快

我们借助puppet-openstack_integration模块的tempest.pp来完成tempest的部署：

```
puppet apply -e 'include openstack_integration::tempest'
```

很快我们就能完成对tempest的部署工作。

核心代码讲解

class tempest

在tempest类中，和其他module不同的一点是关于如何使用源码来安装软件包的技巧。

先说说 `ensure_packages` ，接受列表或哈希类型的package变量并进行安装。以下为使用示例：

- Array类型:

```
ensure_packages(['ksh','openssl'], {'ensure' => 'present'})
```

- Hash类型:

```
ensure_packages({'ksh' => { enure => '20120801-1' } ,  'mypacka
```

ensure_packages和**package**的区别

作用都是一样的，只是在遇到有多个软件包需安装的场景时，ensure_packages使用起来代码更加简洁。

再来看puppet如何下载tempest源码仓库，这里用到了我们在基础模块章节讲到的 `vcsrepo` type：

```
if $git_clone {
  vcsrepo { $tempest_clone_path:
    ensure   => 'present',
    source   => $tempest_repo_uri,
    revision => $tempest_repo_revision,
    provider => 'git',
    require  => Package['git'],
    user     => $tempest_clone_owner,
  }
  Vcsrepo<||> -> Tempest_config<||>
}
```

class tempest同时支持使用venv的方式安装tempest:

```
    if $setup_venv {
      # virtualenv will be installed along with tox
      exec { 'setup-venv':
        command => "/usr/bin/virtualenv ${tempest_clone_path}/.venv
        cwd     => $tempest_clone_path,
        unless  => "/usr/bin/test -d ${tempest_clone_path}/.venv",
        require => [
          Exec['install-tox'],
          Package[$tempest::params::dev_packages],
        ],
      }
      if $git_clone {
        Vcsrepo<||> -> Exec['setup-venv']
      }
    }
```

下述的代码基本上只是使用tempest_config在完成相应服务的配置，就不再展开解释。

小结

在这一节中，我们了解了 `ensure_packages` 函数的使用，也见到 `vcsrepo` 是如何实现下载源码仓库的，也看到如何用Puppet实现python程序的venv安装方式。

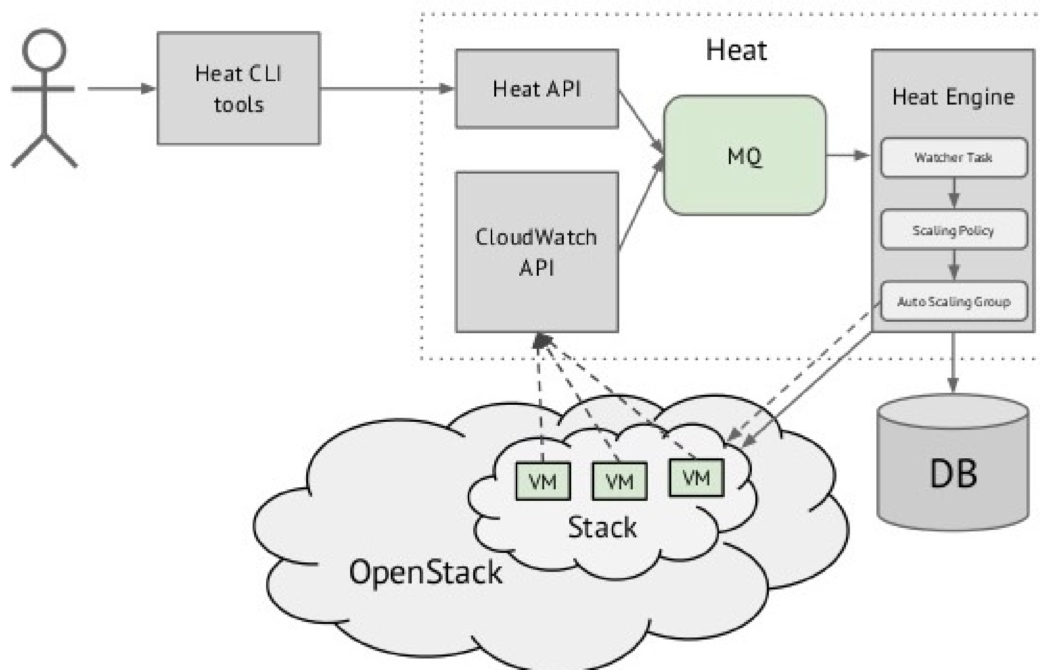
# #

1.部署tempest服务，并开启对sahara，swift的支持 2.对于当前的puppet-tempest，你觉得有什么值得改进的地方？

puppet-heat

本节作者：余兴超
阅读级别：必读
阅读时间: 1小时

AWS CloudFormation服务，为用户提供了编排AWS中的资源的能力。Openstack社区在2012年推出了类似支持编排功能的服务Heat。Heat基本的workflow是这样的：



先睹为快

在了解完Heat后，我们首先通过快速部署一个带有heat所有服务的环境来快速地上手，首先我们需要做一点hack，在 `fixtures/scenario-aio.pp` 文件中追加:

```
include ::openstack_integration::heat
```

然后在终端下执行:

```
   puppet apply -v fixtures/scenario-aio.pp
```

在执行完成后，我们通过systemctl命令，可以看到以下服务并启动：

```
 openstack-heat-api-cfn.service
  openstack-heat-api-cloudwatch.service
  openstack-heat-api.service
  openstack-heat-engine.service
```

先在终端下执行list命令，正常的返回值为空:

```
  openstack stack list
```

接着我们使用一个简单HOT模板来验证一下刚才heat的部署工作是否work，手动创建一个test.yaml文件：

```
heat_template_version: 2015-04-30

description: Simple template to deploy a single compute instance

resources:
  my_instance:
    type: OS::Nova::Server
    properties:
      image: a0a885b4-4045-4dee-bb91-c163c4ba429a
      flavor: m1.nano
```

在终端下输入以下命令：

```
openstack stack create -t test.yaml test
```

观察输出信息中的 `stack_status_reason` ，若为 `started` ，则说明stack创建完成:

```
+---------------------+--------------------------------------------
| Field               | Value
+---------------------+--------------------------------------------
| id                  | 89ccfa5b-31c6-4a86-abe9-72c0a44729f8
| stack_name          | test
| description         | Simple template to deploy a single compute
| creation_time       | 2016-05-31T11:50:39
| updated_time        | None
| stack_status        | CREATE_IN_PROGRESS
| stack_status_reason | Stack CREATE started
+---------------------+--------------------------------------------
```

核心代码讲解

class heat::api

和其他服务类似，heat::api完成了对heat-api软件包，相关配置文件和服务管理。

class heat::api_cfn

和其他服务类似，heat::api_cfn完成了对heat-api-cfn软件包，相关配置文件和服务管理。

class heat::api_cloudwatch

和其他服务类似，heat::api_cloudwatch完成了对heat-api-cloudwatch软件包，相关配置文件和服务管理。

class heat::engine

和其他服务类似，heat::engine完成了对heat-engine软件包，相关配置文件和服务管理。 这里有一个关于 `size` 和 `member` 的用法实例：

- `size` 的作用是返回一个字符串，列表或者哈希的长度。
- `member` 函数则是判断一个变量是否为一个列表的成员。

```
$allowed_sizes = ['16','24','32']
$param_size = size($auth_encryption_key)
if ! (member($allowed_sizes, "${param_size}")) { # lint:ignore:or
  fail("${param_size} is not a correct size for auth_encryption_k
}
```

其他class

- heat::client完成对heatclient软件包的安装配置
- heat::keystone::auh完成heat user,role,endpoint的管理
- heat::keystone::domain完成默认heat domain的创建
- heat::db::mysql完成了heat数据库的创建

小结

Heat服务的架构比较简单，因此在配置上并没有太多复杂的地方，它的核心还是在于结合业务，完成HOT模板的编写。

动手练习

1. 在keystone中添加heat-api-cfn user，service和endpoint
2. 仅部署heat-api和heat-engine服务

- 动手练习

`puppet-swift` 模块

0.基础知识

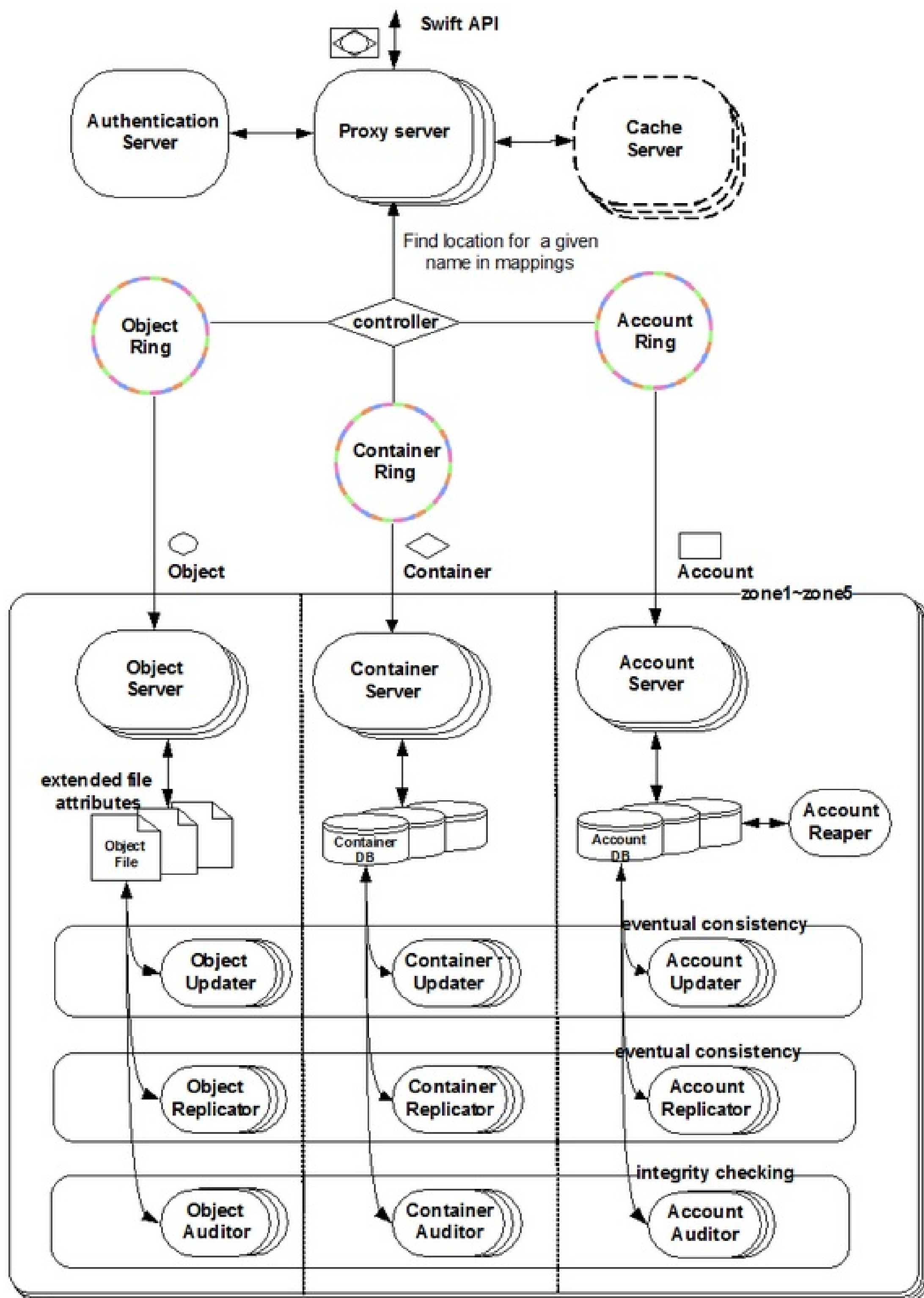
Swift最初源自于Rackspace公司开发的高可用分布式对象存储服务CloudFiles，于2010年贡献给OpenStack开源社区作为最早的核心项目，为用户提供了对象存储，虚机镜像存储，块设备快照存储等诸多功能。Swift可以运行在廉价的标准X86硬件存储上，无需配置RAID（磁盘冗余阵列），通过在软件层面引入一致性哈希算法和数据冗余性，以牺牲一定的数据一致性来最终达到高可用性和可伸缩性，支持多租户模式、容器和对象读写操作，适合解决非结构化数据存储问题。

0.1 Swift基础概念

- 账户（Account）:这里账户不是账号和密码的概念，可以理解为存储区域（Storage area）。
- 容器（container）:容器有自己的metadata，包含了一组object。
- 对象（object）:包含具体数据和metadata。
- 集群（cluster）:表示一个Swift存储集群。
- 区域（region）:表示一个集群中物理隔离的部分。
- 区（zone）:表示物理隔离的节点，可用于控制故障域。
- 节点（node）:表示运行Swift进程的物理服务器。

0.2 Swift组件介绍

Swift服务由众多的组件构成，其架构如下图所示：



名称	说明
swift-proxy-server	对外提供对象服务 API，会根据环的信息来查找服务地址并转发用户请求至相应的账户、容器或者对象服务；由于采用无状态的 REST 请求协议，可以进行横向扩展来均衡负载。
swift-account	提供账户元数据和统计信息，并维护所含容器列表的服务，每个账户的信息被存储在一个 SQLite 数据库中。
swift-container	提供容器元数据和统计信息，并维护所含对象列表的服务，每个容器的信息也存储在一个 SQLite 数据库中。
swift-object	提供对象元数据和内容服务，每个对象的内容会以文件的形式存储在文件系统中，元数据会作为文件属性来存储，建议采用支持扩展属性的 XFS 文件系统。
swift-replicator	会检测本地分区副本和远程副本是否一致，具体是通过对比散列文件和高级水印来完成，发现不一致时会采用推式（Push）更新远程副本，例如对象复制服务会使用远程文件拷贝工具 rsync 来同步；另外一个任务是确保被标记删除的对象从文件系统中移除。
swift-updater	当对象由于高负载的原因而无法立即更新时，任务将会被序列化到在本地文件系统中进行排队，以便服务恢复后进行异步更新；例如成功创建对象后容器服务器没有及时更新对象列表，这个时候容器的更新操作就会进入排队中，更新服务会在系统恢复正常后扫描队列并进行相应的更新处理。
swift-auditor	检查对象，容器和账户的完整性，如果发现比特级的错误，文件将被隔离，并复制其他的副本以覆盖本地损坏的副本；其他类型的错误会被记录到日志中。
swift-account-reaper	移除被标记为删除的账户，删除其所包含的所有容器和对象。
authentication server	验证访问用户的身份信息，并获得一个对象访问令牌（Token），在一定的时间内会一直有效；验证访问令牌的有效性并缓存下来直至过期时间。
cache server	缓存的内容包括对象服务令牌，账户和容器的存在信息，但不会缓存对象本身的数据；缓存服务可采用 Memcached 集群，Swift 会使用一致性散列算法来分配缓存地址。

1.先睹为快

不想看下面大段的代码解析，已经跃跃欲试了？

OK，我们开始吧！

Swift服务比较独立，下面编写learn_swift.pp实现all-in-one部署:

```
$swift_local_net_ip='127.0.0.1'

$swift_shared_secret='changeme'

Exec { logoutput => true }

package { 'curl': ensure => present }


class { '::memcached':
  listen_ip => $swift_local_net_ip,
}

class { '::swift':
  # not sure how I want to deal with this shared secret
  swift_hash_suffix => $swift_shared_secret,
  package_ensure    => latest,
}

# === Configure Storage

class { '::swift::storage':
  storage_local_net_ip => $swift_local_net_ip,
}

# create xfs partitions on a loopback device and mounts them
swift::storage::loopback { '2':
  require => Class['swift'],
}

# sets up storage nodes which is composed of a single
# device that contains an endpoint for an object, account, and cont

swift::storage::node { '2':
  mnt_base_dir         => '/srv/node',
  weight               => 1,
  manage_ring          => true,
```

```
    zone                    => '2',
    storage_local_net_ip => $swift_local_net_ip,
    require                 => Swift::Storage::Loopback[2] ,
  }

  class { '::swift::ringbuilder':
    part_power     => '18',
    replicas       => '1',
    min_part_hours => 1,
    require        => Class['swift'],
  }


  # TODO should I enable swath in the default config?
  class { '::swift::proxy':
    proxy_local_net_ip => $swift_local_net_ip,
    pipeline           => ['healthcheck', 'cache', 'tempauth', 'proxy
    account_autocreate => true,
    require            => Class['swift::ringbuilder'],
  }
  class { ['::swift::proxy::healthcheck', '::swift::proxy::cache']: ]

  class { '::swift::proxy::tempauth':
    account_user_list => [
      {
        'user'    => 'admin',
        'account' => 'admin',
        'key'     => 'admin',
        'groups'  => [ 'admin', 'reseller_admin' ],
      },
      {
        'user'    => 'tester',
        'account' => 'test',
        'key'     => 'testing3',
        'groups'  => [],
      },
    ]
  }
```

在终端执行以下命令:

```
$ puppet apply -v learn_swift.pp
```

等待Puppet命令执行完成，一个单节点的Swift节点部署完成，可以通过swift命令行工具来使用你的对象存储系统了。

2.核心代码讲解

2.1 `class swift`

`swift` 类用于完成以下工作：

- 安装Swift软件包
- 管理swift.conf配置文件
- 管理swift相关目录

2.2 `class swift::proxy`

`swift::proxy` 用于配置swift proxy服务：

```
class swift::proxy(
  $proxy_local_net_ip,
  $port                        = '8080',
  $pipeline                    = ['healthcheck', 'cache', 'tempauth',
  $workers                     = $::processorcount,
  $allow_account_management    = true,
  ...
  $package_ensure              = 'present',
  $service_provider            = $::swift::params::service_provider
) inherits ::swift::params {

  #所有swift_config资源的执行将触发swift-proxy-server的重启
  Swift_config<| |> ~> Service['swift-proxy-server']

  # validate函数用于检查参数的数据类型，若不匹配将抛出异常
  validate_bool($account_autocreate)
```

```
validate_bool($allow_account_management)
validate_array($pipeline)

if($write_affinity_node_count and ! $write_affinity) {
  fail('Usage of write_affinity_node_count requires write_affinit
}

# member函数用于判断右侧参数是否在左侧参数中
if(member($pipeline, 'tempauth')) {
  $auth_type = 'tempauth'
} elsif(member($pipeline, 'swauth')) {
  $auth_type = 'swauth'
} elsif(member($pipeline, 'keystone')) {
  $auth_type = 'keystone'
} else {
  warning('no auth type provided in the pipeline')
}

if(! member($pipeline, 'proxy-server')) {
  warning('pipeline parameter must contain proxy-server')
}

if($auth_type == 'tempauth' and ! $account_autocreate ){
  fail('account_autocreate must be set to true when auth_type is
}

if ($log_udp_port and !$log_udp_host) {
  fail ('log_udp_port requires log_udp_host to be set')
}

package { 'swift-proxy':
  ensure => $package_ensure,
  name   => $::swift::params::proxy_package_name,
  tag    => ['openstack', 'swift-package'],
}

concat { '/etc/swift/proxy-server.conf':
  owner   => 'swift',
  group   => 'swift',
  require => Package['swift-proxy'],
```

```
  }

  $required_classes = split(
    inline_template(
      "<%=
          (@pipeline - ['proxy-server']).collect do |x|
            'swift::proxy::' + x.gsub(/-/){ %q(_) }
          end.join(',')
      %>"), ',')
  concat::fragment { 'swift_proxy':
    target  => '/etc/swift/proxy-server.conf',
    content => template('swift/proxy-server.conf.erb'),
    order   => '00',
    before  => Class[$required_classes],
  }

  Concat['/etc/swift/proxy-server.conf'] -> Swift_proxy_config <||>
  ...
}
```

2.3 class swift::storage

`swift::storage` 完成了配置rsync server的工作，我们在基础模块章节中已介绍过，这里不再赘述。

2.4 `class swift::storage::disk`

`swift::storage::disk` 是一个重要的类，用于管理磁盘分区和文件系统的创建。

puppet``` define swift::storage::disk($base_dir = '/dev', $mnt_base_dir = '/srv/node', $byte_size = '1024', $ext_args = '',) {

include ::swift::deps

if(!defined(File[$mnt_base_dir])) { file { $mnt_base_dir: ensure => directory, owner => 'swift', group => 'swift', require => Anchor['swift::config::begin'], before => Anchor['swift::config::end'], } }

exec { "create_partition_label-${name}": command => "parted -s ${base_dir}/${name} mklabel gpt ${ext_args}", path => ['/usr/bin/', '/sbin','/bin'], onlyif => ["test -b ${base_dir}/${name}","parted ${base_dir}/${name} print|tail -1|grep 'Error'"], before => Anchor['swift::config::end'], }

swift::storage::xfs { $name: device => "${base_dir}/${name}", mnt_base_dir => $mnt_base_dir, byte_size => $byte_size, loopback => false, subscribe => Exec["create_partition_label-${name}"], }

}

```
首先，`swift::storage::disk`会调用"create_partition_label-${name}"exe
也可以通过ext_args参数来传递额外的parted参数。例如，传入'mkpart primary 0%

其次，通过声明`swift::storage::xfs`来指定磁盘xfs文件系统的格式化，并将其挂载

接下来，继续解析`swift::storage::xfs`代码。

### 2.5 swift::storage::xfs
`swift::storage::xfs`完成以下三项功能：

1.完成xfs文件系统的创建：
```puppet
  exec { "mkfs-${name}":
    command => "mkfs.xfs -f -i size=${byte_size} ${target_device}",
    path    => ['/sbin/', '/usr/sbin/'],
    unless  => "xfs_admin -l ${target_device}",
    before  => Anchor['swift::config::end'],
  }
```

2.判断设备的挂载类型
```

```
case $mount_type {
  'path': { $mount_device = $target_device }
  'uuid': { $mount_device = dig44($facts, ['partitions', $target_
            unless $mount_device { fail("Unable to fetch uuid of
          }
  default: { fail("Unsupported mount_type parameter value: '${mou
}
```

3.挂载文件系统

```
swift::storage::mount { $name:
  device       => $mount_device,
  mnt_base_dir => $mnt_base_dir,
  loopback     => $loopback,
}
```

接下来，进入到 `swift::storage::mount` 中。

2.6 swift::storage::mount

`swift::storage::mount` 执行真正的文件系统挂载操作：

1.管理指定的挂载目录

```
file { "${mnt_base_dir}/${name}":
  ensure  => directory,
  owner   => 'swift',
  group   => 'swift',
  require => Anchor['swift::config::begin'],
  before  => Anchor['swift::config::end'],
}
```

2.调用mount资源和exec做双重挂载

```
mount { "${mnt_base_dir}/${name}":
  ensure  => present,
  device  => $device,
  fstype  => $fstype,
  options => "${options},${fsoptions}",
}

# double checks to make sure that things are mounted
exec { "mount_${name}":
  command   => "mount ${mnt_base_dir}/${name}",
  path      => ['/bin'],
  unless    => "grep ${mnt_base_dir}/${name} /etc/mtab",
  # TODO - this needs to be removed when I am done testing
  logoutput => true,
  before    => Anchor['swift::config::end'],
}
```

3.管理挂载目录权限

```
exec { "fix_mount_permissions_${name}":
  command     => "chown -R swift:swift ${mnt_base_dir}/${name}",
  path        => ['/usr/sbin', '/bin'],
  refreshonly => true,
  before      => Anchor['swift::config::end'],
}
```

注意，refreshonly属性表明该资源的执行只能通过被其他资源触发。

2.7 swift::ringbuilder

`swift::ringbuilder` 用于创建account, container以及object的builder文件，并且调用 `swift::ringbuilder::rebalance` 对ring文件进行rebalance操作。

```
class swift::ringbuilder(
  $part_power = undef,
  $replicas = undef,
  $min_part_hours = undef
) {

  include ::swift::deps
  Class['swift'] -> Class['swift::ringbuilder']

  swift::ringbuilder::create{ ['object', 'account', 'container']:
    part_power     => $part_power,
    replicas       => $replicas,
    min_part_hours => $min_part_hours,
  }

  Swift::Ringbuilder::Create['object'] -> Ring_object_device <| |>

  Swift::Ringbuilder::Create['container'] -> Ring_container_device

  Swift::Ringbuilder::Create['account'] -> Ring_account_device <|

  swift::ringbuilder::rebalance{ ['object', 'account', 'container']
```

2.8 swift::ringserver

此类的工作：

- 通过创建一个rsync服务器来启动一个ringdatabase服务

小结

本章咱们介绍了swift的相关概念，通过一个小列子来部署一套all-in-one的swift环境，并且讲解了puppet-swift相关核心代码中的一些小知识点。

动手练习

如何部署一个多节点的swift的集群？要求一个API节点，一个Stoage节点。

puppet-trove

本节作者：廖鹏辉

阅读级别：选读

阅读时间:**2h**

基础知识

Trove 是一个 DBaaS 服务，它能够利用 OpenStack 云平台中的资源，让用户能够以自助的方式快速的构建和管理数据库服务。trove 支持多种数据库，包括关系型和非关系型数据库，例如 MySQL/MariaDB/PostgreSQL/MongoDB/Canssandra/Redis/Couchbase 等等。trove 将这些数据库的管理功能抽象成了统一的接口提供给用户使用。

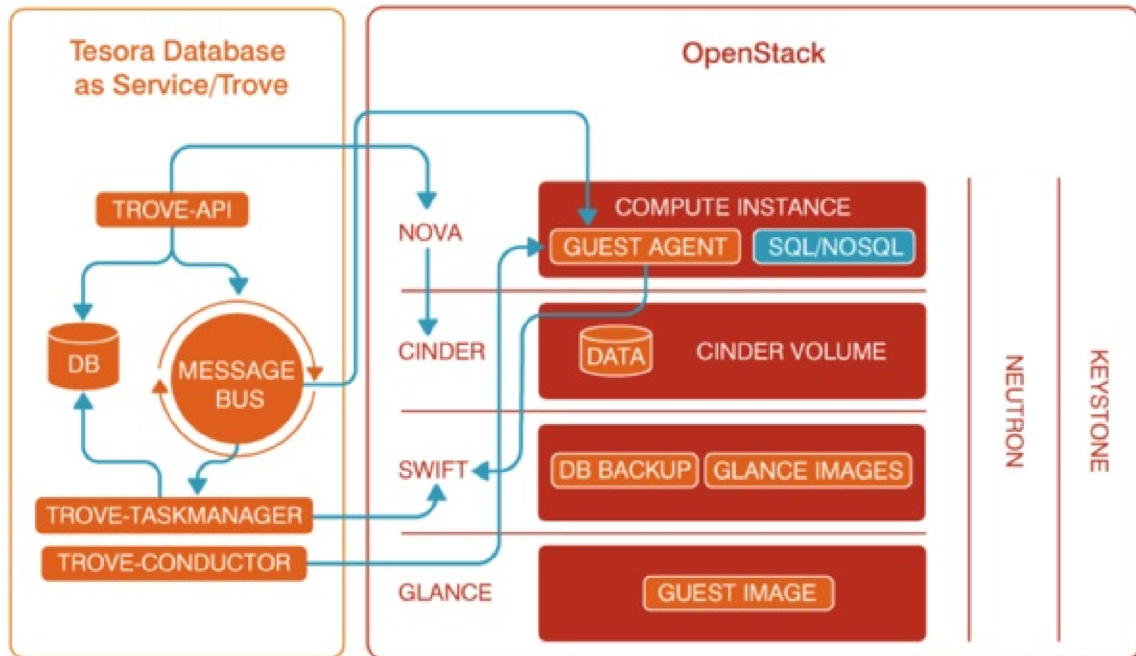
使用 trove 服务能够帮助用户减少数据库的部署，配置和维护成本，能够更轻松的使用各种不同的数据库服务。trove 通过调用 Nova/Cinder/Swift 等其他 OpenStack 服务来进行资源的编排。

trove 服务有下面这些组件：

- trove-api，对外提供 REST-ful 的 API
- trove-taskmanager，监听在消息队列上，负责完成虚拟机实例的启动，以及实例的整个生命周期的管理，并对数据库进行管理操作
- trove-guestagent，运行在虚拟机实例内部，监听在消息队列上，负责对数据库进行管理的运维操作
- trove-conductor，接收虚拟机示例发送的信息，并更新各个实例的状态，它的

作用类似于 nova-conductor

这些组件之间以及与其他 OpenStack 服务之间的关系如下：



先睹为快

部署 trove 服务需要依赖于其他的 OpenStack 组件，因此建议先部署核心的 OpenStack 组件，最后部署 trove 服务。可以使用 puppet-trove 模块的示例代码来部署 trove 服务：

```
puppet apply trove/examples/site.pp
```

代码中使用 `trove::client` 来部署客户端组件，使用 `::trove::api` / `::trove::conductor` / `::trove::taskmanager` 来部署 trove 的各个服务。使用 `::trove::keystone::auth` 类来完成 keystone 相关资源的创建，使用 `::trove::db::mysql` 来完成数据库的创建。

class trove

trove 这个类用于安装 openstack-trove 基础包，同时使用 trove_config 资源对 trove 进行基础配置（例如 nova/neutron/cinder 等服务的访问 URL），并且定义了一些消息队列，数据库相关的参数，供各个服务使用。

例如，下面的代码使用 trove_config 配置了 nova 的访问 URL：

```
if $nova_compute_url {
  trove_config { 'DEFAULT/nova_compute_url': value => $nova_compu
}
else {
  trove_config { 'DEFAULT/nova_compute_url': ensure => absent }
}
```

class trove::api

由于众多 OpenStack 模块都使用 oslo 公共库，这些服务都需要进行 oslo 相关的配置，这些配置在 trove 模块中被抽象成为单独的类，来进行响应的配置， `trove::db` 用于 oslo.db 相关的配置， `trove::logging` 用于 oslo.log 相关的配置

`trove::api` 中，调用了 `trove::db` 来进行数据库配置，调用 `trove::logging` 来进行日志相关的配置。

`trove::api` 还使用了 `puppet-oslo` 模块中的 define 资源来进行 oslo 相关的配置，例如，进行 MQ 相关的配置：

```
    oslo::messaging::rabbit {'trove_config':
      rabbit_hosts          => $::trove::rabbit_hosts,
      rabbit_host           => $::trove::rabbit_host,
      rabbit_port           => $::trove::rabbit_port,
      rabbit_ha_queues      => $::trove::rabbit_ha_queues,
      rabbit_userid         => $::trove::rabbit_userid,
      rabbit_password       => $::trove::rabbit_password,
      rabbit_virtual_host   => $::trove::rabbit_virtual_host,
      rabbit_use_ssl        => $::trove::rabbit_use_ssl,
      kombu_reconnect_delay => $::trove::kombu_reconnect_delay,
      amqp_durable_queues   => $::trove::amqp_durable_queues,
      kombu_ssl_ca_certs    => $::trove::kombu_ssl_ca_certs,
      kombu_ssl_certfile    => $::trove::kombu_ssl_certfile,
      kombu_ssl_keyfile     => $::trove::kombu_ssl_keyfile,
      kombu_ssl_version     => $::trove::kombu_ssl_version
    }
```

关于 puppet-oslo 模块相关的细节可以参考本书对 puppet-oslo 模块的介绍。这里值得注意的是在代码中直接调用了 `trove` 这个类中的变量使用，为什么 `trove::api` 能够使用 `trove` 类中的变量呢，这时因为 `trove::api` 使用 inherits 继承了 `trove` 类，关于 inherits 的用法，可以参考这里。

class trove::conductor

trove-conductor 服务的部署也比较简单，和 trove-api 类似，配置一些基础配置和 oslo 相关配置，就可以启动服务了，在 puppet-trove 模块总，服务的管理都是使用 `trove::generic_service` 这个 define 资源来完成的，例如， trove-conductor 服务的使用：

```
  trove::generic_service { 'conductor':
    enabled        => $enabled,
    manage_service => $manage_service,
    package_name   => $::trove::params::conductor_package_name,
    service_name   => $::trove::params::conductor_service_name,
    ensure_package => $ensure_package,
  }
```

class trove::taskmanager

trove-taskmanager 服务的部署和 trove 其他服务也是类似的，唯一需要注意的是 trove-taskmanager 类中也对 guest-agent 的配置文件进行了管理，管理使用的是模板的方式：

```
file { $guestagent_config_file:
  content => template('trove/trove-guestagent.conf.erb'),
  require => Anchor['trove::install::end'],
}
```

小结

trove 服务的部署比较简单，使用 puppet 能够方便的部署起 trove 服务起来，如果想进一步学习 trove 服务的使用，可以参考 openstack 官方的文档。

动手练习

- 部署 trove 服务，制作一个基于 mysql 的镜像，并导入到 trove 中
- 使用 mysql 镜像启动数据库实例

Puppet-sahara模块介绍

本节作者：付广平

建议阅读时间 **2h**

基础知识

Why Sahara

近年来大数据可谓如火如荼，哪个企业不说搞大数据都要被嘲讽技术落后，程序员不张口闭口MapReduce、Nosql都不敢说自己学计算机的。而谈到大数据就必然提到Hadoop/Spark，似乎谈大数据就等价于说Hadoop/Spark。

如今Hadoop再也不是当初的HDFS、MapReduce、Hbase这么简单了，现在大家谈的Hadoop往往表示的是一个庞大的Hadoop生态圈，包括了YARN、Spark、Sqoop、Hive、Impara、Alluxio等组件。面对如此庞大复杂的分布式系统，面临的首要挑战问题就是如何快速、高效部署和维护。

当面临小规模集群时，我们也许并不需要构建一套复杂的自动化部署工具，只需要从官方下载jar包分发到集群的各个节点，手动一一配置即可完成简单Hadoop集群部署。Hadoop部署非常灵活的同时也造成部署架构复杂，并且一旦规模大时，手动部署方式往往捉襟见肘，调试和维护难度系数直线上升。

面临以上集群部署和维护的诸多痛点问题，很多公司基于社区版本开发了自己的Hadoop产品发行版以及一套完整的自动化部署工具，这些工具不仅能够支持节点自动发现以及可视化部署，还能实时监控集群的健康状态。主流的包括Cloudera公司开发的cloudera-manager工具，支持在Web页面快速部署大规模CDH集群，Hortonworks公司开发的Ambari工具也支持在Web页面上交互来完成HDP集群的部署。这些工具大大简化了部署和监控流程，降低了维护成本。

以上工具虽然很好地完成了Hadoop集群的自动化部署和监控，但其部署工具本身往往还需要手动部署，并且直接构建在物理集群之上，难以实现资源的按需使用以及弹性扩展，也不利于通过云服务的形式快速交付。

Openstack Sahara旨在基于IaaS之上自动化部署Hadoop集群，不仅支持原生Hadoop、Spark、Storm的快速部署，还集成了目前主流的部署工具，比如前面提到的cloudera-manager以及Ambari。也就是说，通过Sahara能够分分钟部署一个CDH或者HDP集群。

不仅如此，Sahara还实现了MapReduce即服务的接口，通过Sahara API能够在Web页面上方便地创建DataSource，然后通过表单上传Jar包即可提交Hadoop Job，使开发者只需要专注于业务开发本身，而不需要关注底层实现，大大提高了开发效率。

Sahara是Openstack的高层服务，构建在Nova、Cinder、Neutron、Heat等服务之上。本章接下来将重点讨论如何在Openstack平台上部署Sahara组件。

Sahara的几个概念

本小节简单介绍下Sahara涉及的几个概念，主要针对感兴趣的读者能够快速了解Sahara，这些内容和部署关系不大，读者可直接跳过本节。

Sahara主要包含以下几个概念:

- Plugin：即Hadoop集群插件，不同的发行版和版本插件不同，如创建CDH集群使用cdh插件、创建Spark使用spark插件等，类似于驱动（driver）的概念。
- Image：Sahara创建集群的每一个节点其实都是一台虚拟机，Image即指定虚

拟机使用的镜像，不同的插件对应的镜像不同，使用前必须和插件绑定，即注册镜像。通常每个插件的镜像都会包含CentOS和Ubuntu两种版本镜像。
- Node Group Template：节点模板，即定义虚拟机模板，主要包含以下内容:
  - 插件：定义使用的Hadoop集群发行版和版本。
  - 资源：该节点使用的Flavor、Availability Zone、volume卷大小、安全组等。
  - 进程组：定义该节点启动什么服务，比如namenode，datanode，spark-master,spark-slave,hue。
  - Hadoop配置参数：比如hdfs_client_java_heapsize,hadoop_job_history_dir等，不同的发行版配置项不同。
- Cluster Template：集群模板，定义集群拓扑和规模大小，集群模板由Node Group Template组合而成，定义一个集群由几个datanode、几个spark-worker构成等，同时还定义Hadoop的一些配置信息，比如HDFS副本数等。
- Cluster：集群实例，已经创建的实例，集群实例由集群模板创建。集群实例还支持扩容和缩容操作，增加或者减少node个数。

Sahara组件介绍

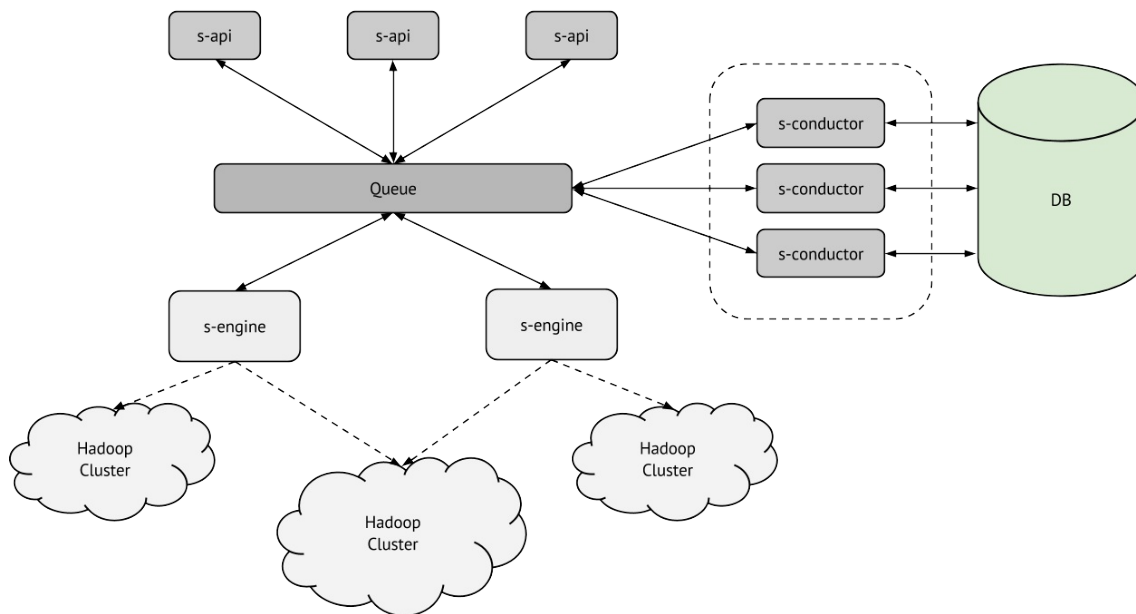
欲知如何部署Sahara，首先需要了解Sahara包含的组件以及模块。和Openstack其它大多数服务一样，Sahara同样需要依赖于消息队列、数据库等基础组件。

最开始Sahara只包含sahara-api一个服务，负责响应用户请求、访问数据库、创建集群等所有工作，造成单服务负载过高并且缺乏HA支持。社区于是提出了下一代架构(https://wiki.openstack.org/wiki/Sahara/NextGenArchitecture)，新架构把sahara-api拆分成两个服务:

- sahara-api: 和大多数Openstack API服务类似，主要为用户提供RESTFul API接口。
- sahara-engine: 负责执行用户的各项任务，包括创建集群和提交用户提交的Job等。

访问数据库也单独分离出了一个独立的模块，称之为sahara-conductor，但注意和nova-conductor不一样，它只是一个模块，而不是一个服务，后期可能会发展成一个独立的服务来接管数据库访问工作。

Sahara官方的新架构图如下:



Sahara服务相对来说还是比较简单的，只包含sahara-api和sahara-engine两个服务。下一小节中将开始介绍sahara的部署问题。

谈谈Sahara部署

前面提到sahara服务相对简单，但不得不说，部署起来却大小坑不计其数。Sahara的工作原理本不应该在这里提及，但在不了解其工作原理的前提下部署Sahara，可以毫不夸张地说: No Way!

由于篇幅有限，对Sahara工作原理感兴趣的读者可以参考官方文档或者阅读源码，本文给出的简化工作流程仅供参考:

- 验证集群。主要检查集群模板是否合法，比如HDFS没有部署namenode、datanode数少于HDFS副本数等都属于不合法的集群。
- 调用Heat创建资源，比如虚拟机、网络、volume、安全组等。
- 通过ssh配置集群并启动集群服务，配置工作包括更新hosts文件、修改Hadoop xml文件等，启动服务如ResourceManager、NodeManager、Datanode、Namenode等等。
- 若集成的是厂商部署工具，则还需要调用其API部署集群，比如调用Cloudera-manager RESTFul API创建集群等。
- 等待所有服务启动完成后，集群创建完成。

从以上步骤可知，**Sahara**部署前必须先部署**Heat**服务，在M版本后这是必需的，M版之前可以使用direct engine，目前已经被彻底废弃。

sahara-engine是通过ssh连接虚拟机完成集群配置的，因此**sahara-engine**必须能够和虚拟机所在网络连通。

目前sahara-engine连通虚拟机的方式有以下几种:

- flat private network，这种方式不支持Neutron网络，不考虑。
- floating IPs，即给所有虚拟机分配公有IP，通过公有IP访问虚拟机。
- network namespace，通过网络命名空间访问虚拟机，sahara-engine必须部署在网络节点，且不支持多网络节点情况(想想为什么？)。
- agent模式，这个尚未实现，主要想参考Trove的agent模式，通过消息队列通信。

以上4种方式其实只有中间两种方式可用，但若集成厂商的Hadoop发行版部署工具并且需要调用厂商工具API部署集群的情况，不支持network namespace模式，因为即使能通过进入netns方式ssh连接虚拟机，也不可能调用虚拟机内部的API服务（除非打通管理网和虚拟机网络）。简而言之，**CDH**和**HDP**不支持**netns**模式。

因此，若要通过**Sahara**部署**CDH**或者**HDP**集群，请使用**floating IPs**模式，并开启虚拟机自动分配**floating ip**功能。

Sahara大多数配置项和其它服务类似，比如日志配置、RabbitMQ配置、认证配置等等，Sahara专有的需要注意的配置项如下(均在`/etc/sahara/sahara.conf`的`DEFAULT`配置组):

- `use_floating_ips`: 若sahara-engine配置使用浮动IP访问虚拟机，则需要设置为`True`，此时建议配置nova配置项`auto_assign_floating_ip`为`True`，否则创建集群时堵塞直到用户手动分配浮动IP。
- `use_neutron`: 使用Neutron网络设置为`True`，否则若使用废弃的`nova-network`设置为`False`。
- `use_namespaces`: 若sahara-engine使用network namespace方式访问虚拟机，需要设置该配置项为`True`。
- `use_rootwrap`: 该配置项需要设置为`True`，否则ssh连接虚拟机时出错。
- `rootwrap_command`: 设置为`"sudo sahara-rootwrap /etc/sahara/rootwrap.conf"`，原因同上。
- `plugins`: 开启的插件列表，N版本前插件列表是静态配置的，N版本后可以动态配置。
- `proxy_command`: sahara-engine使用net ns访问虚拟机时ssh的ProxyCommand参数，默认值为`'ip netns exec ns_for_{network_id}`

`nc {host} {port}'` 。

关于Sahara的高可用，sahara-api由于是HTTP服务，高可用肯定是没有问题的，创建多个实例并放在LB之上即可。

但sahara-engine虽然和nova-conductor、nova-scheduler等服务一样都是消息消费服务，你可以通过部署多实例来提高服务的可用性，但并不能说实现了高可用。Sahara的任务通常都是分阶段的长任务，比如创建一个集群大概需要数分钟时间，但一个任务只能由一个sahara-engine实例全程负责，如果中途挂了，其它sahara-engine实例并不能接管工作。尤其注意在集群扩容操作时，如果**sahara-engine**奔溃了，将导致集群可能永远处于中间状态，甚至导致集群瘫痪。

OK，以上啰嗦了那么多，无非是想引导读者自己想明白Sahara应该怎么部署。Sahara只有两个服务，sahara-api通常部署在控制节点，而sahara-engine部署在哪个节点取决于连接虚拟机使用何种访问模式，相信读者已经有自己的答案了。OK，Let the elephant fly on our Openstack!

先睹为快

前面介绍了Sahara基(ku)础(zao)知识，提到Sahara部署时需要注意的几个问题。我们已经知道若sahara-engine使用浮动ip连接虚拟机，则sahara-engine无所谓部署在哪个节点了，只要保证能够连通虚拟机即可，通常我们会部署在控制节点上。社区为此在puppet-sahara中实现了专门的类同时在一个节点中部署sahara-api和sahara-engine，使一键部署sahara测试环境成为可能:

```
puppet apply -v puppet-sahara/examples/basic.pp
```

以上命令执行完毕，一个单节点Sahara环境就已经部署完成了。

不过正如前面所说，**Sahara**是一个高层服务，依赖于底层**Openstack**基础服务，因此在部署**Sahara**前请务必保证**Keystone**、**Nova**、**Cinder**、**Glance**、**Heat**、**Neutron**等能够正常工作。

核心代码讲解

Openstack服务部署套路

在使用自动化工具部署任何系统之前，首先你得了解需要部署系统的工作原理，并能手动部署之。手动部署过Openstack的一定知道部署Openstack服务的套路，无论你部署Nova，还是部署Cinder、Glance甚至Heat，基本都是这个套路:

1. 创建数据库。
2. 通过Keystone创建用户、服务、endpoint等。
3. 下载必要包。
4. 修改配置，主要包括RabbitMQ、数据库连接等配置。
5. 调用xxxx-manager创建数据库的表。
6. 启动服务。

值得庆幸的是，Sahara部署也完全遵循Openstack服务的部署套路，除了套路里包含的东西，几乎没有其它任何额外新鲜工作。这里就不再重复介绍手动部署过程了，感兴趣的读者可以参考Openstack大数据项目Sahara实践总结。

无论采用何种方式部署，万变不如其中，其实自动化部署工具就是替代我们手动敲的命令，实现步骤是完全一样的，接下来将分析puppet-sahara各个模块实现以及与我们手动部署时对应的步骤如何关联起来的。

数据库配置

首先看 `sahara::db` 这个类，该类位于项目根路径下，主要定义数据库的一些全局通用配置，这些配置是脱离于使用mysql还是postsql的，类定义的原型如下:

```
class sahara::db (
  $database_db_max_retries = $::os_service_default,
  $database_connection      = 'mysql+pymysql://sahara:secrete@localh
  $database_idle_timeout    = $::os_service_default,
  $database_min_pool_size   = $::os_service_default,
  $database_max_pool_size   = $::os_service_default,
  $database_max_retries     = $::os_service_default,
  $database_retry_interval = $::os_service_default,
  $database_max_overflow    = $::os_service_default,
)
```

接下来在 `sahara::db::mysql` 是专门针对使用mysql数据库的配置，源码如下:

```
class sahara::db::mysql(
  $password,
  $dbname        = 'sahara',
  $user          = 'sahara',
  $host          = '127.0.0.1',
  $allowed_hosts = undef,
  $charset       = 'utf8',
  $collate       = 'utf8_general_ci',
) {

  validate_string($password)

  ::openstacklib::db::mysql{ 'sahara':
    user          => $user,
    password_hash => mysql_password($password),
    dbname        => $dbname,
    host          => $host,
    charset       => $charset,
    collate       => $collate,
    allowed_hosts => $allowed_hosts,
  }

  ::Openstacklib::Db::Mysql['sahara'] ~> Exec<| title == 'sahara-dt
}
```

以上相当于调用 `::openstacklib::db::mysql` 创建数据库，对应部署套路第一条。

最后我们通知执行 `Exec<| title == 'sahara-dbmanage'` ，这其实就相对于对应套路第5条，源代码为:

```
class sahara::db::sync(
  $extra_params = '--config-file /etc/sahara/sahara.conf',
) {

  include ::sahara::params

  Package <| tag == 'sahara-package' |> ~> Exec['sahara-dbmanage']
  Exec['sahara-dbmanage'] ~> Service <| tag == 'sahara-service' |>

  Sahara_config <||> -> Exec['sahara-dbmanage']
  Sahara_config <| title == 'database/connection' |> ~> Exec['sahar

  exec { 'sahara-dbmanage':
    command     => "sahara-db-manage ${extra_params} upgrade head",
    path        => '/usr/bin',
    user        => 'sahara',
    refreshonly => true,
    try_sleep   => 5,
    tries       => 10,
    logoutput   => on_failure,
    tag         => 'openstack-db',
  }

}
```

以上几个类共同协作完成了Sahara数据库表的初始化。

Sahara服务认证配置

认证配置对应部署套路第2步，主要包括调用Keystone API创建sahara用户、服务、endpoint等，puppet-sahara代码实现在 `sahara::keystone::auth` ，该类的实现和前面几个服务非常类似，如Nova、Cinder等，再次不再重复，有兴趣的读者可以直接阅读源码。

Sahara配置管理

Sahara配置文件主要包括 `/etc/sahara/sahara.conf` 和 `/etc/sahara/api-paste.ini` 两个文件，其中 `/etc/sahara/sahara.conf` 绝大多数配置项由 `init.pp` 下的 `sahara` 类管理，使用形如 `xyz/key:value` 的键值对保存,其中 `xyz` 表示所在的配置组， `key` 表示配置项名称，后面的 `value` 是配置项的值，样例如下:

```
sahara_config {
    'DEFAULT/plugins':                value => join(any2array($plugins)
    'DEFAULT/use_neutron':            value => $use_neutron;
    'DEFAULT/use_floating_ips':       value => $use_floating_ips;
    'DEFAULT/host':                   value => $host;
    'DEFAULT/port':                   value => $port;
    'DEFAULT/default_ntp_server':     value => $default_ntp_server;
  }
```

除了 `sahara` 类中定义的配置参数，在 `sahara::config` 中可定义一些额外配置项，通常通过hieradata定义。

另外除了基本配置外，和大多数其它服务一样，还需要配置policy，对应类为 `sahara::policy` ，该类实现和其它服务类似，这里不再重复介绍。

该步骤对应手动部署套路第4条。

Sahara服务安装和启动

前面我们已经知道Sahara由sahara-api和sahara-engine两个服务，分别对应的类为 `sahara::service::api` 和 `sahara::service::engine` ，这两个类都定义了包的安装、服务配置等。以api服务为例，其核心代码为:

```
class sahara::service::api (
  $api_workers    = $::os_workers,
  $enabled        = true,
  $manage_service = true,
  $package_ensure = 'present',
) {

  include ::sahara::policy
  include ::sahara::params

  Sahara_config<||> ~> Service['sahara-api']
  Class['sahara::policy'] ~> Service['sahara-api']

  package { 'sahara-api':
    ensure => $package_ensure,
    name   => $::sahara::params::api_package_name,
    tag    => ['openstack', 'sahara-package'],
    notify => Service['sahara-api'],
  }
  service { 'sahara-api':
    ensure     => $service_ensure,
    name       => $::sahara::params::api_service_name,
    enable     => $enabled,
    hasstatus  => true,
    hasrestart => true,
    require    => Package['sahara-api'],
    tag        => 'sahara-service',
  }

}
```

以上可以很清晰地从代码看出，该类就是对应部署套路的第3条和第6条。

小结

在这里，我们介绍了puppet-sahara的核心代码实现以及各个模块完成的工作，通过和手动部署套路联系在一起，相信读者能更容易理解代码的原理。当然该module还有许多重要的class我们并没有涉及，例如：`sahara::logging`，`sahara::nofity`等等。这些就留给读者自己去阅读代码了，当然在后期的版本中，我也会进一步去完善puppet-sahara的核心代码内容。

动手练习

1. 使用puppet-sahara部署Sahara，要求sahara-engine支持`net_ns`访问模式。
2. 想想为什么sahara-engine使用`net_ns`访问虚拟机不支持多网络节点情况。

puppet-manila

本节作者：周维宇

阅读级别：选读

阅读时间:**2h**

基础知识

manila 是一个 "Shared Filesystems as a service” 服务，通过driver不同的后端共享存储系统来给提供共享文件存储

manila 服务有下面这些组件：

- manila-api，对外提供 REST-ful 的 API
- manila-scheduler，根据预定的策略选择合适的manila-share节点来处理用户请求
- manila-share，通过driver处理实际的创建创建共享卷等请求

先睹为快

部署 manila 服务需要依赖于其他的 OpenStack 组件，因此建议先部署核心的 OpenStack 组件，最后部署 manila 服务。另外由于我们选用nfs作为存储后端，所以你要先部署一个nfs server。

```
# 请根据你的实际部署情况修改参数
class { 'manila':
  sql_connection  => 'mysql://manila:secret_manila_password@opens
  rpc_backend     => 'rabbit',
  rabbit_password => 'secret_rpc_password_for_manila',
  rabbit_host     => 'openstack-controller.example.com',
  verbose         => true,
}

class {'manila::api':
  keystone_password  => $keystone_password,
  keystone_auth_host => $keystone_auth_host,
  os_region_name     => 'DEFAULT'
}
class {'manila::scheduler':
  scheduler_driver => 'manila.scheduler.filter_scheduler.FilterSc
}
class {'::manila::share':
  package_ensure => $package_ensure
}
manila::backend::glusternfs {'nfs':
  glusterfs_target              => [remoteuser@]<volserver>:/<vol
  glusterfs_mount_point_base    => '/nfs',
  glusterfs_nfs_server_type     => 'Gluster',
  glusterfs_path_to_private_key => 'ssh_private_key_path',
  glusterfs_ganesha_server_ip,  => 'ganesha_server_ip',
```

核心代码讲解

class manila

manila 这个类用于安装 openstack-manila 基础包，同时使用 manila_config来管理日志/消息队列/SSL等参数

例如，下面的代码使用 manila_config 配置了SSL相关的参数：

```
# SSL Options
if $use_ssl {
  manila_config {
    'DEFAULT/ssl_cert_file' : value => $cert_file;
    'DEFAULT/ssl_key_file' :  value => $key_file;
  }
  if $ca_file {
    manila_config { 'DEFAULT/ssl_ca_file' :
      value => $ca_file,
    }
  } else {
    manila_config { 'DEFAULT/ssl_ca_file' :
      ensure => absent,
    }
  }
} else {
  manila_config {
    'DEFAULT/ssl_cert_file' : ensure => absent;
    'DEFAULT/ssl_key_file' :  ensure => absent;
    'DEFAULT/ssl_ca_file' :   ensure => absent;
  }
}
```

class manila::db

调用 `manila_config` 来进行数据库相关的配置,比较有意思的是下面这段代码

```
validate_re($database_connection_real,
  '^(sqlite|mysql(\+pymysql)?|postgresql):\/\/(\S+:\S+@\S+\/\S+)?

# 根据不同的数据库后端来执行不同的操作
if $database_connection_real {
  case $database_connection_real {
    /^mysql(\+pymysql)?:\/\//: {
      require 'mysql::bindings'
      require 'mysql::bindings::python'
      if $database_connection_real =~ /^mysql\+pymysql/ {
        $backend_package = $::manila::params::pymysql_package_nar
      } else {
        $backend_package = false
      }
    }
    /^postgresql:\/\//: {
      $backend_package = false
      require 'postgresql::lib::python'
    }
    /^sqlite:\/\//: {
      $backend_package = $::manila::params::sqlite_package_name
    }
    default: {
      fail('Unsupported backend configured')
    }
  }
```

class manila::api

除了传统的装软件包/改配置/启动服务三板斧,没有别的好讲的

class manila::scheduler

同上

class manila::share

同上

class manila::backends

配置开启哪些存储后端

```
class manila::backends (
  $enabled_share_backends = undef
) {

  # Maybe this could be extented to dynamicly find the enabled name
  manila_config {
    'DEFAULT/enabled_share_backends': value => join($enabled_share_
  }

}
```

define manila::backend::glusternfs

```
  # 通过manila_config来修改manila配置
  manila_config {
    "${share_backend_name}/share_backend_name":            value =>
    "${share_backend_name}/share_driver":                  value =>
    "${share_backend_name}/glusterfs_target":              value =>
    "${share_backend_name}/glusterfs_mount_point_base":    value =>
    "${share_backend_name}/glusterfs_nfs_server_type":     value =>
    "${share_backend_name}/glusterfs_path_to_private_key": value =>
    "${share_backend_name}/glusterfs_ganesha_server_ip":   value =>
  }
```

小结

manila 服务的部署比较简单，使用 puppet 能够方便的部署起 manila 服务起来，如果想进一步学习 manila 服务的使用，可以参考 openstack 官方的文档。

动手练习

- 部署 manila 服务，创建两台云主机和一个共享卷并挂载

puppet-rally

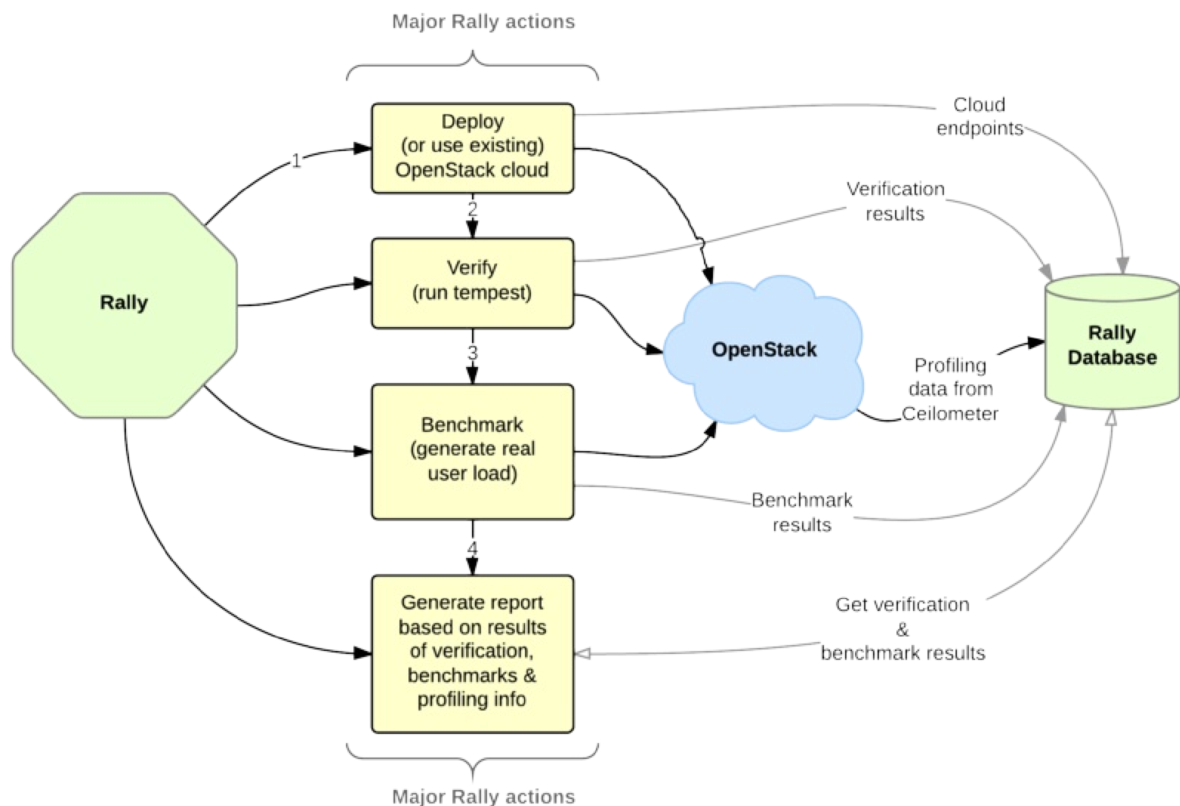
本节作者：余兴超

阅读级别：选读

阅读时间: **40**分钟

基础知识

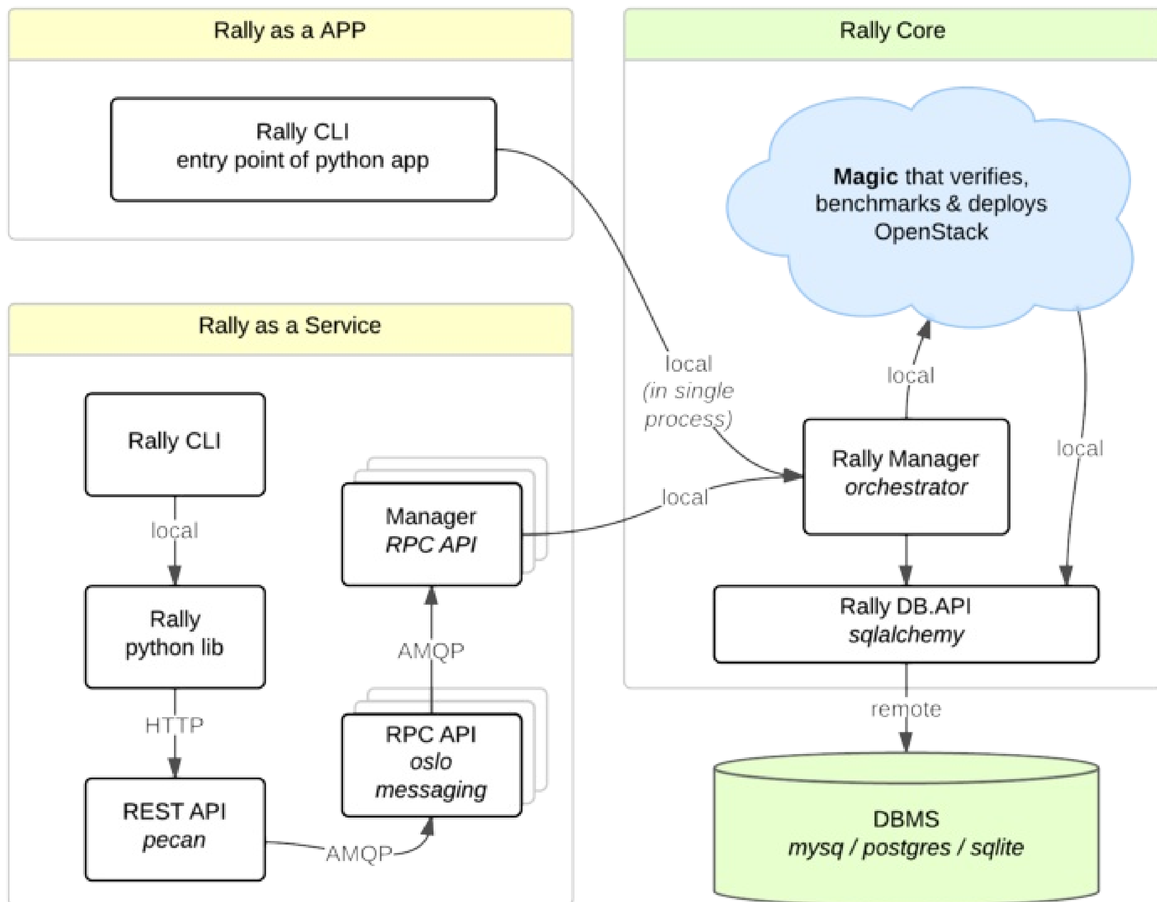
`Rally` 项目是Openstack性能测试服务，可以被用于Openstack CI/CD中的基本工具链中，以提高Openstack的SLA。下图给出了Rally与Deployment,Verify,Benchmark之间的关系以及其执行流程。不过Rally当前的主要工作仍然集中在benchmark上，社区的进度比较缓慢。



架构简介

Openstack大多数项目属于as-a-service类型，因此Rally提供了service和CLI两种方式：

- Rally as-a-Service 以web service方式对外提供服务
- Rally as-an-App 作为轻量级命令行工具使用



先睹为快

> puppet-rally模块目前没有使用Release机制管理，请使用master分支代码

在终端下执行以下命令:

```
puppet apply -e 'include rally'
```

然后就可以开始使用rally了，是不是so easy？

核心代码讲解

`puppet-rally` 模块中，我们主要介绍 `class rally` 和 `class rally::settings` ：

class rally

```
include ::rally::db    #配置数据库
include ::rally::logging  #配置日志
include ::rally::settings #rally.conf配置文件

# Keep backward compatibility
$openstack_client_http_timeout_real = pick($::rally::settings::op

# rally软件包的安装
package { 'rally':
  ensure => $ensure_package,
  name   => $::rally::params::package_name,
  tag    => ['openstack', 'rally-package'],
}
# 是否清理非Puppet管理的配置
resources { 'rally_config':
  purge => $purge_config,
```

class rally::settings

rally配置文件中涉及到各个服务的参数设置，以cinder为例，在[benchmark]下就有以下参数：

- $volume_create_poll_interval
- $volume_create_prepoll_delay
- $volume_create_timeout
- $volume_delete_poll_interval
- $volume_delete_timeout

在 `puppet-rally` 模块下，将各服务的参数设置，拆为单独的class，放置在settings/目录下，统一被 `rally::settings` 调用：

```
class rally::settings (
  $project_domain                = $::os_service_default,
  $resource_deletion_timeout     = $::os_service_default,
  $resource_management_workers   = $::os_service_default,
  $user_domain                   = $::os_service_default,
  $openstack_client_http_timeout = undef,
) {

  #管理rally各服务的配置
  include ::rally::settings::cinder
  include ::rally::settings::ec2
  include ::rally::settings::glance
  include ::rally::settings::heat
  include ::rally::settings::ironic
  include ::rally::settings::manila
  include ::rally::settings::murano
  include ::rally::settings::nova
  include ::rally::settings::sahara
  include ::rally::settings::swift
  # 此类等待https://review.openstack.org/#/c/337412/被Merge
  include ::rally::settings::tempest

  rally_config {
    'cleanup/resource_deletion_timeout':         value => $resource
    'users_context/project_domain':              value => $project_
    'users_context/resource_management_workers': value => $resource
    'users_context/user_domain':                 value => $user_dor
  }
}
```

小结

本节简要介绍了Rally服务以及如何使用 `puppet-rally` 模块部署Rally服务。当前，Rally服务作为Openstack平台的性能测试项目，目前的使用场景还是比较有限的，因为当前Rally只能支持API级别的性能测试，并且多数API的后端操作是异步

的，如果只关注API的响应结果，意义不大。不过值得庆幸的是，Rally项目有来自国内两家公司的core reviewer（Kun Huang和Li Yingjun），我们期待着Newton版本中Rally的发展和变化。

动手练习

1. 设置glance_image_create_timeout为60s
2. 使用MySQL替换默认的SQlite作为数据库后端
3. 当前`puppet-rally`能否支持从源码安装Rally？需要如何修改？

puppet-designate

本节作者：薛飞扬

阅读级别：选读

建议阅读时间 **2**小时

DNS

想要搞懂Designate项目，没有正确的DNS姿势怎么行？所以先别急，我们先来聊一聊DNS的那些事。

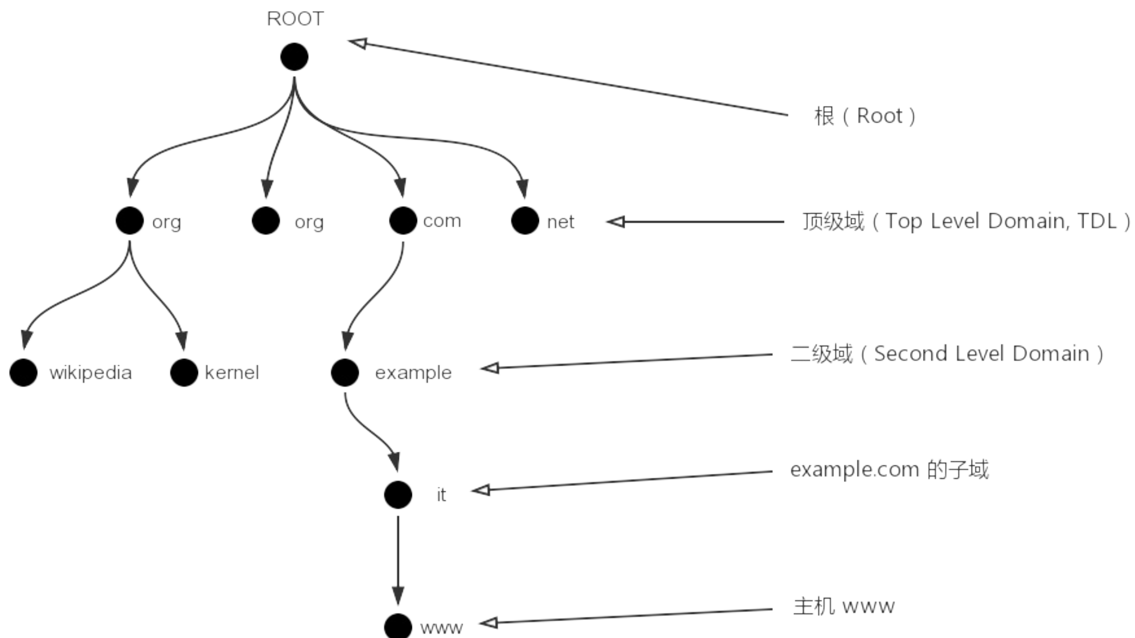
DNS简介

DNS的全称是Domain Name System，负责主机名和互联网络地址之间的映射。在我们上网或者发送电子邮件的时候，一般都会使用主机名而不是IP地址，因为前者便于我们记忆，但是对于计算机来讲，TCP/IP网络中要求每一个互连的计算机都具有其唯一的IP地址，并基于这个IP地址进行通信。DNS能够帮助我们将主机名转换成具体的IP地址。从而完成主机之间的通信。

DNS层级结构

DNS是一个层级的分布式的数据库，以C/S架构工作，它将互联网名称（域名）和IP地址的对应关系记录下来，可以为客户端提供名称解析的功能。它允许对整个数据库的各个部分进行本地控制，借助备份和缓存机制，DNS将具有足够的强壮性。

DNS数据库以层级的树状结构组织，最顶级的服务器被称为「根」（root），以 . 表示，它是所有子树的根。root将自己划分为多个子域（subdomain），这些子域包括com，net，org，gov，net等等，这些子域被称为顶级域（Top Level Domain, TDL）。再进一步，各顶级域再将自己划分成多个子域，子域还可以在划分子域，最后树的叶子节点就是某个域的主机名。整个结构如下图所示：



每个域的名称服务器仅负责本域内的主机的名称解析，如果需要解析子域的主机，就需要再向其子域的名称服务器查询。这样一来，无论主机在哪个域内，都可以从根开始一级一级的找到负责解析此主机名称的域，然后完成域名解析。

BIND DNS 服务器

BIND是由Berkely大学研发的一款开源DNS服务器程序，它是目前世界上使用最为广泛的DNS服务器软件,支持各种unix平台和windows平台。在CentOS系统中，由bind软件包提供安装。

Bind的两个配置文件：

- /etc/named.conf：主要规范主机的设定、zone file 的所在、权限的设定等；主要配置如下：

```
options {
        listen-on port 53 { 127.0.0.1; };  //定义DNS监听在哪个IP的特定
        directory "/var/named";  //指定DNS区域文件存放目录
        dump-file "/var/named/data/cache_dump.db";  //默认服务器存放数
        statistics-file "/var/named/data/named_stats.txt";  //默认统
        memstatistics-file "/var/named/data/named_mem_stats.txt";
        allow-query { any; };  //定义允许哪些主机可以查询本地的DNS服务
        recursion no;  //定义是否允许DNS服务器做递归查询
};
```

- 正反解资料库档案(zone file)：/var/named/目录下，一个zone file由多条资源记录组成。

```
$TTL 1D
@       IN SOA  LinuxMaster.test.com.  admin.test.com. (
                                        2016092605      ; serial
                                        21600           ; refresh
                                        3600            ; retry
                                        604800          ; expire
                                        86400 )         ; minimum
        IN NS   LinuxMaster
```

DNS服务器类型

服务器类型	作用
缓存服务器	不负责解析，仅为加速，不需要注册
主DNS服务器	负责解析本地客户端请求
辅助DNS服务器	辅助服务器的区域数据都是从主服务器复制而来，其数据都是只读的

缓存服务器需要配置：

```
options {
        forward only;                     //所有请求转发到forwarders列
        forwarders {
                8.8.8.8;8.8.4.4;          //定义转发请求目的IP
        };
};
```

Master DNS服务器需要配置：

```
zone "test.com" IN {
        type master;
        file "test.com.zone";
};
```

Slave DNS服务器需要配置：

```
zone "test.com" IN {
        type slave;
        masters {ip;};
        file "slaves/test.com.zone";
};
```

Slave必须要与Master相互搭配，当要修改一条记录时，只要手动更改Master那部机器的zone file，重新启动BIND这个服务后（或者等待一定时间），slave会自动同步这条更改的记录。 基本上，不论Master 还是Slave 的资料库，都会有一个代表该资料库新旧的『序号』，这个序号数值的大小，会影响是否要更新的动作。至于更新的方式主要有两种：

- Master主动告知：例如在Master在修改了资料库内容，并且加大资料库序号后，重新启动DNS服务，那master会主动告知slave来更新资料库，此时就能够达成资料同步；
- 由Slave主动提出要求：基本上， Slave会定时的向Master察看资料库的序号，当发现Master资料库的序号比Slave自己的序号还要大(代表比较新)，那么Slave就会开始更新。如果序号不变，那么就判断资料库没有更动，因此不会进行同步更新。

DNS资源记录类型

资源记录： 标准的资源记录具有其基本格式：[name]　　　[ttl]　　　IN
type　　　rdata

类型	含义
IN	此字段用于将当前记录标识为一个INTERNET的DNS资源记录
type	类型字段，用于标识当前资源记录的类型
rdata	用于描述资源的信息且长度可变的必要字段，随CLASS和TYPE的变化而变化

每个区域数据库文件都是由资源记录构成的。type的值主要有：SOA记录、NS记录、A记录、CNAME记录、MX记录和PTR记录。

资源记录类型	描述	一句话描述
起始授权结构（SOA）	用于一个区域的开始，SOA记录后的所有信息均是用于控制这个区域的，每个区域数据库文件都必须包含一个SOA记录，并且必须是其中的第一个资源记录，用以标识DNS服务器管理的起始位置，SOA说明能解析这个区域的dns服务器中哪个是主服务器。	指出当前区域内谁是主DNS服务器
主机（A）	即是A记录，也称为主机记录，是DNS名称到IP地址的映射，用于正向解析。	将域名FQND映射到IP 正向解析
别名（CNAME）	CNAME记录，也是别名记录，用于定义A记录的别名。	将A记录指向的域名指向另外一个域名
邮件交换器（MX）	邮件交换器记录，用于告知邮件服务器进程将邮件发送到指定的另一台邮件服务器。（该服务器知道如何将邮件传送到最终目的地）。	指出当前区域内SMTP邮件服务器IP
名称服务器（NS）	NS记录，用于标识区域的DNS服务器，即是说负责此DNS区域的权威名称服务器，用哪一台DNS服务器来解析该区域。一个区域有可能有多条ns记录，例如zz.com有可能有一个主服务器和多个辅助服务器。	指出当前区域内有几个DNS服务器在提供服务
反向解析(PRT)	是IP地址到DNS名称的映射，用于反向解析。	将IP解析为域名FQDN

DNS客户端

dig是Linux下常用的DNS查询工具，在CentOS系统中，由bind-utils软件包提供，它的使用方法为：

dig -t RRT NAME [@NAME_SERVER]

其中，RRT表示资源记录类型，NAME表示查询的地址，@NAME_SERVER可以指定DNS服务器，如果不指定则使用操作系统默认的DNS服务器。

例如，查询www.kernel.org的IP地址：

```
dig -t A www.kernel.org @114.114.114.114

; <<>> DiG 9.8.3-P1 <<>> -t A www.kernel.org @114.114.114.114
;; global options: +cmd
;; Got answer:
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 55431
;; flags: qr rd ra; QUERY: 1, ANSWER: 4, AUTHORITY: 0, ADDITIONAL:

;; QUESTION SECTION:
;www.kernel.org.            IN    A

;; ANSWER SECTION:
www.kernel.org.        36    IN    CNAME    pub.all.kernel.org.
pub.all.kernel.org.    36    IN    A    149.20.4.69
pub.all.kernel.org.    36    IN    A    198.145.20.140
pub.all.kernel.org.    36    IN    A    199.204.44.194

;; Query time: 28 msec
;; SERVER: 114.114.114.114#53(114.114.114.114)
;; WHEN: Tue Feb  7 11:15:11 2017
;; MSG SIZE  rcvd: 102
```

基础知识

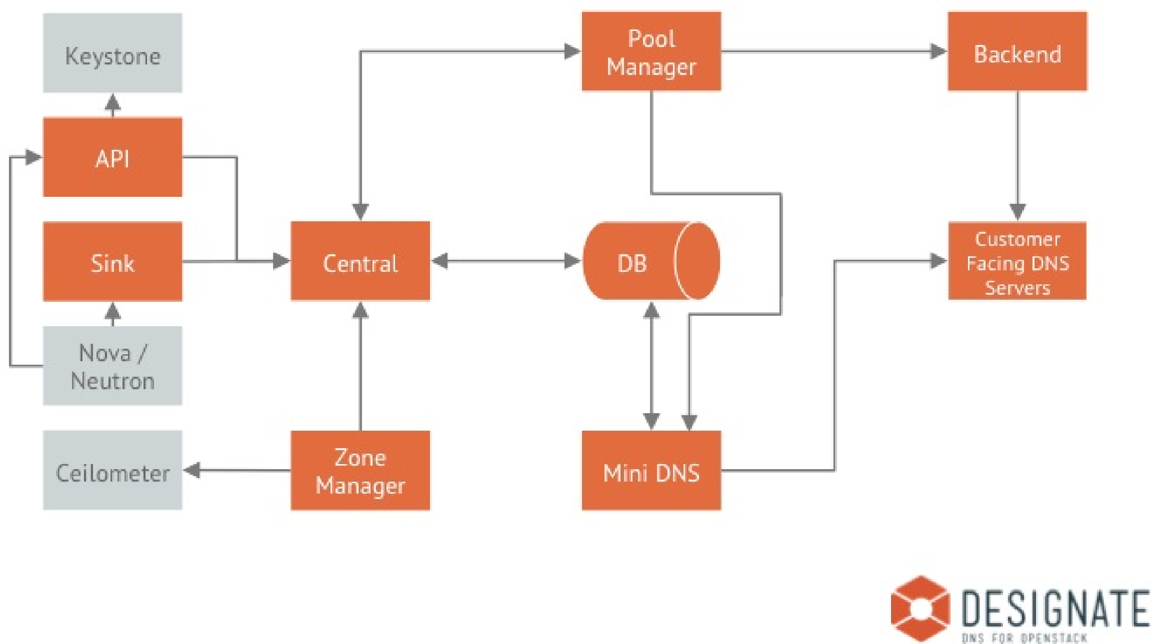
Designate介绍

Designate是OpenStack的DNSaaS组件，它为OpenStack提供了以下服务：

- 域和备案管理的REST API
- 多租户
- 集成了Keystone认证
- 在框架中整合了Nova和Neutron通知（自动生成相应记录）

- 支持PowerDNS和Bind9开箱

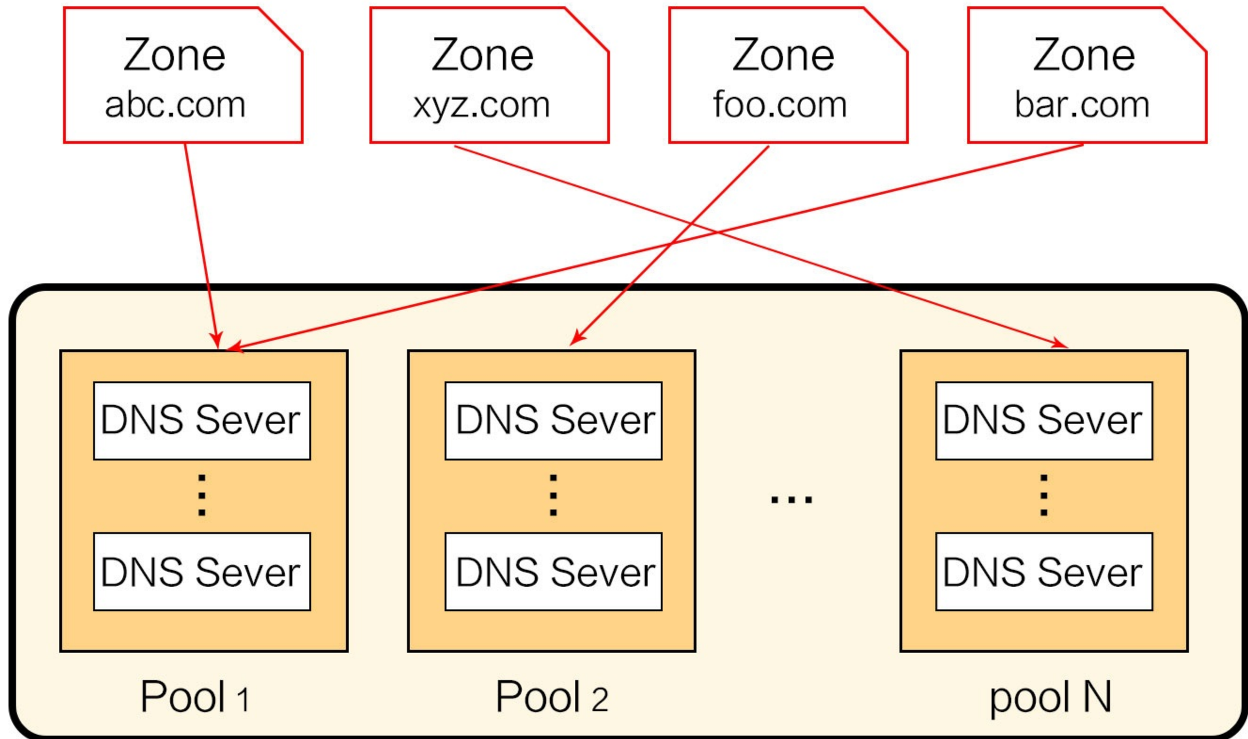
Designate的架构图如下：



包含的服务	简介
designate-api	接收来自远端用户的HTTP/HTTPS请求，通过Keystone验证远端用户的合法性，将HTTP/HTTPS请求传递给Central模块。
designate-central	业务逻辑处理核心。响应API请求以及处理Sink所监听到的来自Nova和Neutron的特定通知事件。同时会存取数据库，对业务逻辑处理所产生的数据进行持久化存储。
designate-mdns	实现了标准的DNS Notify和Zone Transfer的处理。
designate-pool-manager	连接后端驱动，管理DNS服务器池，与MiniDNS配合同步DNS服务器的域名以及资源记录等数据。
designate-sink	监听来自Nova和Neutron的某些事件，用于自动生成域名资源记录，比如当监听到Nova的compute.instance.create.end事件通知后，自动创建一条对应于刚创建的实例的A记录；当监听到Nuetron的floatingip.update.end事件通知后，自动更新一条相应的A记录。

DNS服务器的池划管理

Designate kilo版本所引入的pool manager机制将DNS服务器群划分成多个服务器池（pool），如下图所示，每个服务器池可以配置包含1台或多台DNS服务器。而且，池中的DNS服务器选型还可以不同，也就是说在一个服务器池中，可以有1台BIND服务器，还可以有1台PowerDNS服务器，这是完全支持的。



服务器池的引入目的：

1. 细化域名托管的颗粒度。用户请求托管的域名可以委派到某一个服务器池，而不需要在所有服务器上管理用户的域名和资源记录，降低了管理和运维的复杂度。例如，abc.com委派给pool 1的DNS服务器来管理，xyz.com委派到pool N的DNS服务器来管理。
2. 每个服务器池可以包含多台DNS服务器，实现了高可用性和冗余备份。
3. 服务器池的划分不受地域的限制，可以将分布在不同地域的DNS服务器划归到同一个池中，通过GLB和anycast路由技术可以实现就近DNS查询和负载均衡，加快DNS查询速度。

官方文档上给了一个多个池的使用场景：

The idea is that we'll configure our pools to support different usage levels. We'll define a gold and standard level and put zones in each based on the tenant.

Our gold level will provide 6 nameservers that users have access to where our standard will only provide 2. Both pools will have one master target we write to. 即通过配置不同的池来支持不同的用户等级。

黄金等级的池提供六个nameservers ，而标准等级的池只提供两个。

在mitaka版本，实现了CLI的方法来更新池,即通过创建一个yaml文件来定义池,然后通过desigante-manage来更新池，在后面的designate原理部分，贴出了一个yaml，仅供参考。

先睹为快

在讲解designate模块之前让我们先使用puppet把我们的实验环境部署起来,请根据你的具体环境修改learn_designate.pp

```
include '::rabbitmq'

include '::mysql::server'

#创建database和user
class {'::designate::db::mysql':

  password => $designate_db_password,

}
#创建designate group 和user,安装openstack-designate-common包,修改配置
class {'::designate':

  rabbit_host     => $rabbit_host,

  rabbit_userid   => $rabbit_userid,

  rabbit_password => $rabbit_password,

}

include '::designate::dns'

 #配置designate的后端DNS服务器为bind9，安装bind9包，运行named服务，更改rr

include '::designate::backend::bind9'
```

```
  class {'::designate::db':

    database_connection => "mysql://designate:${designate_db_passwor

  }

  #populate designate database

  include '::designate::db::sync'

 #下面四个class对应desigante的api,central,mdns,pool-manager四个服务

  class {'::designate::api':

    auth_strategy => $auth_strategy,

  }

  class {'::designate::central':

    backend_driver => $backend_driver,

  }

  include '::designate::mdns'

  class {'::designate::pool_manager':

    pool_id => $pool_id,

  }
```

在终端执行以下命令:

```
puppet apply -v learn_designate.pp
```

ok，接下来快创建一个domain试试吧。

```
designate domain-create --name example.com. --email root@example.co

designate domain-list
```

核心代码讲解

backend

Designate backend支持如BIND，PowerDNS等多种类型的DNS服务器，下面只以BIND为例来讲解

Designate backend如果使用应用最为广泛的bind，designate会利用RNDC指令来管理DNS伺服器，所以需要更改rndc的相关配置

```
class designate::backend::bind9 (

 $rndc_host = '127.0.0.1',

 $rndc_port = '953',

 $rndc_config_file = '/etc/rndc.conf',

 $rndc_key_file = '/etc/rndc.key'

) {

 include ::designate
#安装bind相关的包，更改bind相关配置项
 include ::dns

#配置rndc监听的host,port,config和key的目录
 designate_config {

 'backend:bind9/rndc_host' : value => $rndc_host;

 'backend:bind9/rndc_port' : value => $rndc_port;
```

```
  'backend:bind9/rndc_config_file' : value => $rndc_config_file;

  'backend:bind9/rndc_key_file' : value => $rndc_key_file;

  }


 #更改named.conf(或者named.options)文件，允许创建新的zone
  concat::fragment { 'dns allow-new-zones':

  target => $::dns::optionspath,

  content => 'allow-new-zones yes;',

  order => '20',

  }

 }
```

puppet-designate的安装简单来说做了三件事：

- 后端DNS服务器的安装和配置，这一点上面已经有过讲解
- designate相关软件包的安装

```
package { 'designate-common':

 ensure => $package_ensure,

 name => $common_package_name,

 tag => ['openstack', 'designate-package'],

 }

designate::generic_service { 'api':

 enabled => $enabled,

 manage_service => $service_ensure,

 ensure_package => $package_ensure,

 package_name => $api_package_name,

 service_name => $::designate::params::api_service_name,

 }
...
```

- desingate配置文件的管理 除了权限的相关配置，其它的配置项都在/etc/designate/designate.conf中 [oslo_messaging_rabbit]下面是rabbitmq的相关参数，由class designate管理：

```
designate_config {
  'oslo_messaging_rabbit/rabbit_userid' : value => $rabbit_userid;
  'oslo_messaging_rabbit/rabbit_password' : value => $rabbit_passwor
  'oslo_messaging_rabbit/rabbit_virtual_host' : value => $rabbit_vir
  'oslo_messaging_rabbit/rabbit_use_ssl' : value => $rabbit_use_ssl;
  'oslo_messaging_rabbit/kombu_ssl_ca_certs' : value => $kombu_ssl_c
  'oslo_messaging_rabbit/kombu_ssl_certfile' : value => $kombu_ssl_c
  'oslo_messaging_rabbit/kombu_ssl_keyfile' : value => $kombu_ssl_ke
  'oslo_messaging_rabbit/kombu_ssl_version' : value => $kombu_ssl_ve
  'oslo_messaging_rabbit/kombu_reconnect_delay' : value => $kombu_re
  }
```

〔service:api〕 [service:central] [serivce:mdns] [service:pool_manager] 中的配置项分别由class designate::api designate::central designate::mdns designate::manager管理

designate原理

此处以具体的环境为例来介绍designate的原理 实验环境：

- 系统为centos7.2
- designate-api, designate-central, designate-pool-manager, designate-mdns, rabbitmq, mysql, keystone 均部署在10.0.2.250
- bind分别部署在10.0.2.250 10.0.2.249

下面贴出pools.yaml的配置：

```
- also_notifies:
  - host: 10.0.2.249
    port: 53
  attributes: {}
  description: Pool built from configuration on localhost
  id: 794ccc2c-d751-44fe-b57f-8894c9f5c842
  nameservers:
  - host: 10.0.2.250
    port: 53
  - host: 10.0.2.249
    port: 53
  ns_records:
  - hostname: server-250.2.stage.polex.io.
    priority: 1
  - hostname: server-249.2.stage.polex.io.
    priority: 2
  targets:
  - masters:
    - host: 10.0.2.250
      port: 5354
    options:
      rndc_config_file: /etc/rndc.conf
      rndc_host: 127.0.0.1
      rndc_key_file: /etc/rndc.key
      rndc_port: '953'
    type: bind9
  - masters:
    - host: 10.0.2.250
      port: 5354
    options:
      rndc_config_file: /etc/rndc.conf
      rndc_host: 10.0.2.249
      rndc_key_file: /etc/rndc.key
      rndc_port: '953'
    type: bind9
```

Designate工作流程：

- 用户请求designate-api,添加record或者domain

- designate-api发送请求至mq中
- designate-central接收到mq请求,写入db,同时通过mq触发pool_manager进行更新操作
- pool_manager通过rndc(addzone/delzone/notifyzone)三个操作来通知pool_targets中定义的bind来进行操作
- bind使用axfr来请求同步mdns
- mdns从数据库中读取相应的domain信息来响应axfr请求

Target vs. Nameserver

当通过designate 增加／修改／删除记录时，会通过target 去write changes。
当dns客户端去查询记录时，则会通过nameserver.
以本次实验环境为例，当通过designate创建一个名为example.com的domain时，按照上面的pool.yaml配置，相当于执行了这两条命令：

```
rndc -s 127.0.0.1 -p 953 -c /etc/rndc.conf -k /etc/rndc.key addzone
rndc -s 10.0.2.249 -p 953 -c /etc/rndc.conf -k /etc/rndc.key addzon
```

可以看到这些配置项都是在target中定义的。 一个要解析的域名就是一个zone,一个zone对应/var/named/目录下的一个zone_file.在250和249机器上，都能找到新创建的这个文件：

```
[root@server-250.2.stage.polex.io named ]$ ll
total 32
-rw-r--r--. 1 named named  544 Nov  3 17:50 3bf305731dd26307.nzf
drwxrwx---. 3 named named   70 Nov  3 17:35 data
drwxrwx---. 2 named named   58 Nov  4 16:32 dynamic
-rw-r-----. 1 root  named 2076 Jan 28  2013 named.ca
-rw-r-----. 1 root  named  152 Dec 15  2009 named.empty
-rw-r-----. 1 root  named  152 Jun 21  2007 named.localhost
-rw-r-----. 1 root  named  168 Dec 15  2009 named.loopback
drwxr-x---. 2 root  named    6 Oct 19 14:03 puppetstore
-rw-r--r--. 1 named named  409 Nov  4 19:12 slave.example.com.75c9e
drwxrwx---. 2 named named    6 Mar 16  2016 slaves
```

通过dig 查询创建的server1.example.com记录时，返回信息为：

```
[root@server-250.2.stage.polex.io ~ ]$ dig server1.example.com @10.

; <<>> DiG 9.9.4-RedHat-9.9.4-29.el7_2.3 <<>> server1.example.com @
;; global options: +cmd
;; Got answer:
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 31922
;; flags: qr aa rd ra; QUERY: 1, ANSWER: 1, AUTHORITY: 2, ADDITIONA

;; OPT PSEUDOSECTION:
; EDNS: version: 0, flags:; udp: 4096
;; QUESTION SECTION:
;server1.example.com.        IN    A

;; ANSWER SECTION:
server1.example.com.    3600    IN    A    1.2.3.4

;; AUTHORITY SECTION:
example.com.        3600    IN    NS    server-249.2.stage.polex.io
example.com.        3600    IN    NS    server-250.2.stage.polex.io

;; Query time: 0 msec
;; SERVER: 10.0.2.250#53(10.0.2.250)
;; WHEN: Fri Nov 04 18:45:55 CST 2016
;; MSG SIZE  rcvd: 130
```

Hidden Master

前面有提到过，当我们创建一个名为example.com的domain时，执行的rndc命令都指定master host：10.0.2.250， port:5354 在designate.conf的定义中，我们可以看到：

```
[service:mdns]
threads = 1000
host = 0.0.0.0
port = 5354
tcp_backlog = 100
tcp_recv_timeout = 0.5
query_enforce_tsig = False
```

5354是service designate-mdns监听的端口。 所以说，250，249两台机器上的bind都是使用axfr来请求同步mdns，它们的记录都是同步过来的（所以某种意义上讲，它们都是slave节点）。 这样的好处就是，如果250，249有公网ip,它的53接口能被访问，那么它的记录是不能通过外网来更改的（外网只能查询），对于dns记录的更改只能通过内网（10.0.2.250）designate api的方式。

使用场景

解析私有域名

```
#创建一个domain
designate domain-create --name example03.com. --email root@example.
#创建一条记录
designate record-create --name server1.example03.com. --type A --da
#使用dig测试
[root@server-250.2.stage.polex.io named ]$ dig @10.0.2.250 server1.

; <<>> DiG 9.9.4-RedHat-9.9.4-29.el7_2.3 <<>> @10.0.2.250 server1.e
; (1 server found)
;; global options: +cmd
;; Got answer:
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 51643
;; flags: qr aa rd ra; QUERY: 1, ANSWER: 1, AUTHORITY: 2, ADDITIONA

;; OPT PSEUDOSECTION:
; EDNS: version: 0, flags:; udp: 4096
;; QUESTION SECTION:
;server1.example03.com.        IN    A

;; ANSWER SECTION:
server1.example03.com.    3600    IN    A    1.2.3.4

;; AUTHORITY SECTION:
example03.com.        3600    IN    NS    server-250.2.stage.polex.
example03.com.        3600    IN    NS    server-249.2.stage.polex.

;; Query time: 0 msec
;; SERVER: 10.0.2.250#53(10.0.2.250)
;; WHEN: Fri Nov 04 19:20:58 CST 2016
;; MSG SIZE  rcvd: 132
```

与nova和neutron集成

designate-sink 通过nova handler 和 neutron handler，自动生成域名资源记录，比如当监听到Nova的compute.instance.create.end事件通知后，自动创建一条对应于刚创建的实例的A记录；当监听到Nuetron的floatingip.update.end事件通知后，自动更新一条相应的A记录。 一个domain如下：

```
designate record-list 98775b46-c868-4dd8-94fd-6bd789a5dbaa
+--------------------------------------+------+-----------------+--
| id                                   | type | name            | c
+--------------------------------------+------+-----------------+--
| e61e814f-5bd4-4de8-9c11-5c87fe1b2799 | SOA  | bluesky.edu.au. | h
+--------------------------------------+------+-----------------+--
```

当创建一个虚拟机后，自动创建一条A记录：

```
designate record-list 98775b46-c868-4dd8-94fd-6bd789a5dbaa
+--------------------------------------+------+--------------------
| id                                   | type | name
+--------------------------------------+------+--------------------
| 680eb3f5-016f-45a3-ac34-e79ff46bf8df | SOA  | bluesky.edu.au.
| c66481bd-bd4f-4f90-976a-67b6654c4c60 | A    | phobos.bluesky.edu.
+--------------------------------------+------+--------------------
```

小结

puppet-designate需要配置的文件仅有designate.conf，为了方便管理与配置，puppet把使用到的四个服务都分别写为了一个.pp的类，这样也方便我们管理这些配置项。这里讲解到功能没有涉及到跟nova，neutron的集成，集成之后的效果是当创建虚拟机或创建浮动IP后，创建的虚拟机或浮动ip的A记录记录会自动同步相应的zone 中 ，有兴趣的同学可以在官网查看。

动手练习

1.部署分布式的backend后端，并使用不同的DNS Server（BIND,PowerDNS,MysqlBIND）作存储后端。

2.手动配置多个pool,不同的pool管理的各自的DNS Server(通过创建yaml文件，使用designate-manage命令实现)。

3.安装designate-sink服务，修改nova.conf和neutron.conf相应配置，创建虚拟机或floating ip ，观察designate record的变化。

puppet-aodh模块

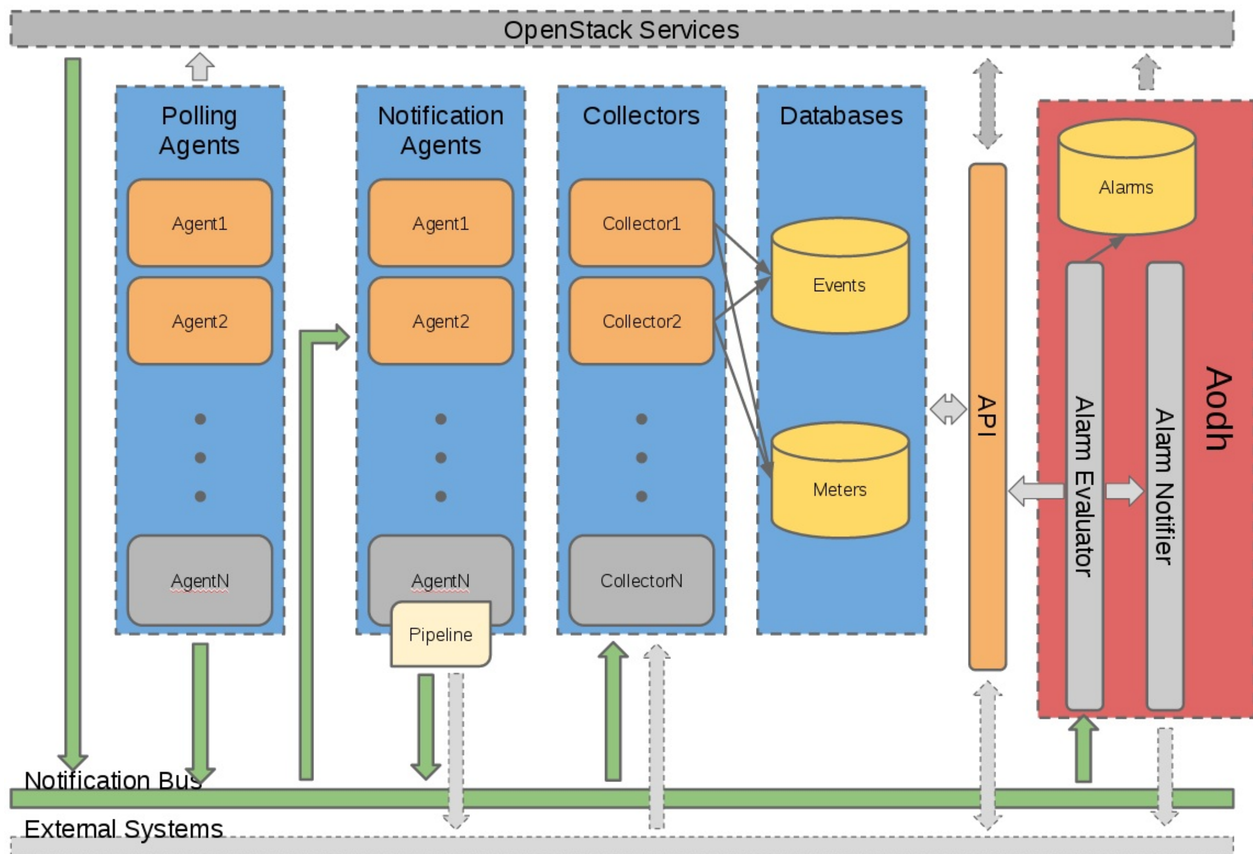
0.理解Aodh

Aodh是Openstack告警项目，最初在Havana版本中作为Ceilometer项目的一个组件(ceilometer-alarm)出现在Ceilometer项目中，在Liberty版本中演变成了独立项目Aodh，用户可以为独立事件或者样本设置阈值和告警机制。

Aodh服务由以下组件组成：

名称	说明
openstack-aodh-api	为告警数据的存储和访问提供接口
openstack-aodh-evaluator	根据统计的数据，来评估是否需要触发告警
openstack-aodh-notifier	根据配置的告警方式，发出告警
openstack-aodh-listener	监听事件，触发事件相关的告警

各个组件之间的关系如下图所示:



1.先睹为快

不想看下面大段的代码解析，已经跃跃欲试了？

OK，我们开始吧！

打开虚拟机终端并输入以下命令：

```
$ puppet apply examples/aodh.pp
```

等待命令执行完成，Puppet完成了对Aodh服务的安装。

注：部署Aodh服务，依赖于Keystone服务。

2.核心代码讲解

`class aodh`

`class aodh` 完成了以下三项任务:

- Aodh common包的安装
- Aodh配置文件的清理
- RabbitMQ和AMQP选项的管理

其中rabbit和AMQP相关的选项管理均是通过oslo::messaging::rabbit和oslo::messaging::amqp来管理，关于puppet-oslo模块，将会在下一个章节详细介绍。

```
  oslo::messaging::rabbit { 'aodh_config':
    rabbit_userid               => $rabbit_userid,
    rabbit_password             => $rabbit_password,
    rabbit_virtual_host         => $rabbit_virtual_host,
    rabbit_host                 => $rabbit_host,
    rabbit_port                 => $rabbit_port,
    rabbit_hosts                => $rabbit_hosts,
    rabbit_ha_queues            => $rabbit_ha_queues,
    heartbeat_timeout_threshold => $rabbit_heartbeat_timeout_thresh
    heartbeat_rate              => $rabbit_heartbeat_rate,
    rabbit_use_ssl              => $rabbit_use_ssl,
    kombu_reconnect_delay       => $kombu_reconnect_delay,
    kombu_ssl_version           => $kombu_ssl_version,
    kombu_ssl_keyfile           => $kombu_ssl_keyfile,
    kombu_ssl_certfile          => $kombu_ssl_certfile,
    kombu_ssl_ca_certs          => $kombu_ssl_ca_certs,
    kombu_compression           => $kombu_compression,
    amqp_durable_queues         => $amqp_durable_queues,
  }
```

在package资源中，有一个元属性tag:

```
package { 'aodh':
  ensure => $package_ensure_real,
  name   => $::aodh::params::common_package_name,
  tag    => ['openstack', 'aodh-package'],
}
```

`tag` 顾名思义就是标签，资源、类和定义都可以对其标记，一个资源可以有任意数量的标记。有多种标记资源的方式，以上代码是使用了元参数tag，对aodh package资源 添加了2个tag：'openstack','aodh-package'。这些tag会在 `aodh::deps` 中使用，用于收集标记为 `aodh-package` 的package资源:

```
anchor { 'aodh::install::begin': }
 -> Package<| tag == 'aodh-package'|>
 ~> anchor { 'aodh::install::end': }
```

class aodh::api

api的主要是提供数据的接口，为告警数据的提供存储和访问。在class aodh::api中先是定义了以下几个依赖关系：

```
  if $auth_strategy == 'keystone' {
    include ::aodh::keystone::authtoken
  }

  Aodh_config<||> ~> Service[$service_name]
  Class['aodh::policy'] ~> Service[$service_name]

  Package['aodh-api'] -> Service[$service_name]
  Package['aodh-api'] -> Service['aodh-api']
  Package['aodh-api'] -> Class['aodh::policy']
  package { 'aodh-api':
    ensure => $package_ensure,
    name   => $::aodh::params::api_package_name,
    tag    => ['openstack', 'aodh-package'],
  }
```

代码中两种符号'->'和'~>'，这两者都是描述资源间的依赖，前面已经介绍过。同时在模块中都同样使用keystone作为认证。API类中其余代码则是对参数进行配置，略过。

class aodh::evaluator

```
  if $manage_service {
    if $enabled {
      $service_ensure = 'running'
    } else {
      $service_ensure = 'stopped'
    }
  }
  Package['aodh'] -> Service['aodh-evaluator']
  service { 'aodh-evaluator':
    ensure     => $service_ensure,
    name       => $::aodh::params::evaluator_service_name,
    enable     => $enabled,
    hasstatus  => true,
    hasrestart => true,
    tag        => ['aodh-service','aodh-db-sync-service']
  }
  )
```

aodh-evaluator 服务的部署和 aodh-api 类似，配置一些基础配置和 oslo 相关配置，就可以启动服务了。

class aodh::notifier

```
  Package['aodh'] -> Service['aodh-notifier']
  service { 'aodh-notifier':
    ensure     => $service_ensure,
    name       => $::aodh::params::notifier_service_name,
    enable     => $enabled,
    hasstatus  => true,
    hasrestart => true,
    tag        => 'aodh-service',
  }
}
```

class aodh::listener

```
  Package['aodh'] -> Service['aodh-listener']
  service { 'aodh-listener':
    ensure     => $service_ensure,
    name       => $::aodh::params::listener_service_name,
    enable     => $enabled,
    hasstatus  => true,
    hasrestart => true,
    tag        => 'aodh-service',
  }
}
```

小结

从上述的代码中，咱们可清晰看到aodh的安装、数据库创建与同步、认证、api、evaluator、notifier、listener服务的配置、启动、管理。源于aodh手动部署文档。

动手练习

1. 配置Aodh运行在httpd下运行
2. 使用AMQP替换RabbitMQ

第五章 PuppetOpenStack公共库和工具类模块

公共库类型模块统一管理者各个服务的公共配置，例如puppet-oslo，管理着各个oslo库的配置；另外一类是工具类模块，它们的主要作用是提供一个模块的基础配置，测试配置，同步配置等与模块本身相关：

- Common Puppet library (OpenStackLib)
- Common Ruby helper library (puppet-openstack_spec_helper)
- Puppet OpenStack helpers (OpenStackExtras)
- Virtual Bridging (OpenvSwitch)
- Integration CI tools (Puppet OpenStack Integration)
- Blueprints (Puppet OpenStack Specs)
- Compliant tool (Cookiebutter)
- Sync tool (Modulesync)
- Oslo libraries (Oslo)

puppet-oslo

本节作者：余兴超

建议阅读时间 **30**分钟

这是读者和作者都会感到轻松又欢快的一章，因为puppet-oslo模块的结构非常简单。 回到正题，puppet-oslo模块是我(xingchao)在16年年初提出并和鹏辉一起贡献到社区的puppet module。它的目的就是为了消灭当时存在于各模块中大量的冗余代码，例如：每个模块当中都有rabbitmq的配置，都有log的配置，都有db的配置，那么为何不做一个公共库，把这些代码抽取出来呢？

先睹为快

很可惜，这是一个公共define库，不会单独存在，而是被其他模块调用来使用。

核心代码讲解

所有代码的结构都是一样的，就是针对某个oslo.xxx库的参数设置，因此在这里我们只举一个例子来说明：

define oslo::log

在oslo::log的代码中，继续给大家讲解一些函数的使用。

以下代码中，我们看到了：

- is_service_default用于判断变量是否使用了默认值；
- validate_hash用于判断变量是否为hash类型；
- join使用分隔符将list连接成字符串;

- sort将字符串和数组进行按单词排序；
- join_keys_to_values将key和value使用分隔符连接，例如：join_keys_to_values({'a'=>1,'b'=>2}, " is ")` 结果为 ["a is 1","b is 2”]

```
if is_service_default($default_log_levels) {
  $default_log_levels_real = $default_log_levels
} else {
  validate_hash($default_log_levels)
  $default_log_levels_real = join(sort(join_keys_to_values($defau
}
```

我们接着往下看，下面的关键是create_resources函数，终于到了值得讲一讲的地方了。

Puppet中的迭代用法

学习Puppet的人常会问起Puppet中的迭代用法，因为多数Puppet用户都有命令式编程的经验，比如说在Bash Shell下，使用for语句来表示循环。但是Puppet是一门声明式DSL(domain-specific language)，DSL不是图灵完备的(Turing complete)。因此在Puppet 4.x之前(Language: Iteration and Loops)，是不支持迭代语法的，不过从Puppet 3.3开始，可以通过一定的配置来开启Puppet中的试验性迭代功能。

我们还有另外一种方式来实现迭代功能，那就是使用create_resources函数，create_resource可以接受3个参数:

- resource名称
- hash类型变量
- 可选，hash变量，用于设置resrouce公共属性

```
# A hash of user resources:
$myusers = {
  'nick' => { uid    => '1330',
              gid    => allstaff,
              groups => ['developers', 'operations', 'release'], },
  'dan'  => { uid    => '1308',
              gid    => allstaff,
              groups => ['developers', 'prosvc', 'release'], },
}
create_resources(user, $myusers)

$defaults = {
  'ensure'   => present,
  'provider' => 'ldap',
}
create_resources(user, $myusers, $defaults)
```

OK，我们再回过头来看这段代码，是不是就很容易理解了？

```
$log_options = {
  'DEFAULT/debug'                          => { value => $debug },
  'DEFAULT/verbose'                        => { value => $verbose
  'DEFAULT/log_config_append'              => { value => $log_conf
  'DEFAULT/log_date_format'                => { value => $log_date
  'DEFAULT/log_file'                       => { value => $log_file
  'DEFAULT/log_dir'                        => { value => $log_dir
  'DEFAULT/watch_log_file'                 => { value => $watch_lo
  'DEFAULT/use_syslog'                     => { value => $use_sysl
  'DEFAULT/syslog_log_facility'            => { value => $syslog_l
  'DEFAULT/use_stderr'                     => { value => $use_stde
  'DEFAULT/logging_context_format_string'  => { value => $logging_
  'DEFAULT/logging_default_format_string'  => { value => $logging_
  'DEFAULT/logging_debug_format_suffix'    => { value => $logging_
  'DEFAULT/logging_exception_prefix'       => { value => $logging_
  'DEFAULT/logging_user_identity_format'   => { value => $logging_
  'DEFAULT/default_log_levels'             => { value => $default_
  'DEFAULT/publish_errors'                 => { value => $publish_
  'DEFAULT/instance_format'                => { value => $instance
  'DEFAULT/instance_uuid_format'           => { value => $instance
  'DEFAULT/fatal_deprecations'             => { value => $fatal_de
}

create_resources($name, $log_options)
```

小结

这章的内容比较简单，我们主要介绍了几个函数的使用说明，着重说明了Puppet中的迭代，它们的加入使得代码逻辑变得更加强大。

动手练习

1. define和class有什么区别？为什么要使用define而不使用class？

`puppet-vswitch` 模块

Open vSwitch(OVS)是一个高质量的、多层虚拟交换机，使用开源Apache2.0许可协议。它的目的是让大规模网络自动化可以通过编程扩展,同时仍然支持标准的管理接口和协议，Open vSwitch支持多种linux 虚拟化技术，包括Xen/XenServer，KVM和irtualBox。

`puppet-vswitch` 项目是由OpenStack社区维护的模块，用于配置和管理Openvswitch。

`puppet-vswitch` 项目地址: https://github.com/openstack/puppet-vswitch

在OVS中，有三个非常重要的基本概念：

- Bridge: 表示一个以太网交换机，其功能是根据流规则，把从端口收到的数据包转发到一个或多个端口
- Port: 收发数据包的单元，每个Port都属于一个特定的bridge
- Interface: 连接到Port的网络接口设备，可以是物理网卡，也可以是虚拟网卡

1.先睹为快

不想看下面大段的代码说明，已经跃跃欲试了？

OK，我们开始吧！

打开虚拟机终端并输入以下命令：

```
$ puppet apply -e 'class {'vswitch': provider => 'ovs'}'
```

等待命令执行完成，Puppet完成了Openvswitch安装并启动了ovs服务。

2.代码讲解

2.1 class vswitch

`class vswitch` 的逻辑比较简单，使用include函数声明了"::vswitch::${provider}"。

```
class vswitch (
  $provider = $vswitch::params::provider
) {
  $cls = "::vswitch::${provider}"
  include $cls
}
```

2.2 class vswitch::ovs

`class vswitch::ovs` 用于管理Openvswitch的软件包和服务,管理服务的代码如下：

```
    'Redhat': {
      service { 'openvswitch':
        ensure => true,
        enable => true,
        name   => $::vswitch::params::ovs_service_name,
      }
    }
```

管理openvswitch软件包的代码如下，指定安装软件的顺序在启动服务之前:

```
  package { $::vswitch::params::ovs_package_name:
    ensure => $package_ensure,
    before => Service['openvswitch'],
  }
```

自定义资源类型vs_port/vs_bridge/vs_config

`puppet-vswitch` 模块提供了vs_port和vs_bridge两个自定义资源类型，分别用于管理port和bridge。

例1, 使用 `vs_bridge` 创建一个名为br-ex的ovs bridge：

```
vs_bridge { 'br-ex':
  ensure => present,
}
```

例2，使用vs_port将端口eth1绑定到br-ex上：

```
vs_port { 'eth1':
  ensure => present,
  bridge => 'br-ex',
}
```

例3，使用vs_config添加新配置项到Openvswitch配置文件中：

```
vs_config { 'parameter_name':
  ensure => present,
  value => "some_value"
}
```

在 `vswitch::ovs` 指定了资源的执行顺序，vs_bridge和vs_port在openvswitch服务之后。

```
  Service['openvswitch'] -> Vs_port<||>
  Service['openvswitch'] -> Vs_bridge<||>
```

3.动手练习

1. 创建一个vs bridge br-tun, 并且把eth1加入到br-tun

puppet-openstacklib

本节作者：廖鹏辉

建议阅读时间 **45**分钟

在部署一个 OpenStack 集群时，我们可能需要安装多个 OpenStack 项目，由于 OpenStack 项目都是按照类似的设计模式开发的，因此这些不同的服务的架构都具有某些共同的特点，例如每个服务一般都会有一个专用的数据库，都会使用消息队列来完成内部组件通信，一般都由 Python 开发，可以使用 WSGI 的方式进行部署等等...

由于这些服务的共同特性，在部署 OpenStack 服务时，我们的很多操作往往需要重复进行，例如为每个服务创建数据库，以及数据库的用户和访问权限，这些操作可能存在于每个服务对应的 Puppet 模块中，为了尽可能的减少重复性代码，社区将一些常用的通用性操作写成 Puppet 中的 define 并放在一个公共模块中供其他模块使用，这样其他模块只需要调用这个公共模块中定义好 define 资源即可。这个公共模块就是 `puppet-oepnstacklib` ，它的作用类似于软件开发中的公共类库。

核心代码讲解

`puppet-openstacklib` 这个模块主要提供了下面这些资源：

- `openstacklib::service_validation` ，用于执行脚本或命令对服务可用性进行验证
- `openstacklib::db::mysql` ，用于完成 mysql 数据库，数据库用户的创建和用户的授权

- `openstacklib::db::postgresql` ，用于完成 postgresql 数据库，数据库用户的创建和用户的授权
- `openstacklib::policy::base` ，用于配置 policy.json 文件
- `$::os_service_default` 这个 facter，用于设置 openstack 配置文件的默认值
- 用于定义各个服务配置的自定义资源所用的基础类

openstacklib::db::mysql

这里以 `puppet-nova` 模块为例，来看看 openstack 模块如何去使用 `puppet-openstacklib` 模块中的资源。例如，在配置数据库用户，权限以及数据库的创建时，nova 模块中的 nova::db::mysql 使用了 `openstacklib::db::mysql` 来创建数据库，用户，以及对用户授权：

```
::openstacklib::db::mysql { 'nova':
  user          => $user,
  password_hash => mysql_password($password),
  dbname        => $dbname,
  host          => $host,
  charset       => $charset,
  collate       => $collate,
  allowed_hosts => $allowed_hosts,
}
```

openstacklib::service::validation

nova 模块中的 nova::api 类调用了 `openstacklib::service::validation` 这个资源，用户可以自己定义服务的检查脚本，来对服务进行健康检查：

```
  if $validate {
    $defaults = {
      'nova-api' => {
        'command'  => "nova --os-auth-url ${auth_uri} --os-tenant-r
      }
    }
    $validation_options_hash = merge ($defaults, $validation_optior
    create_resources('openstacklib::service_validation', $validatic
  }
```

openstacklib::policy::base

如果想自己定义 policy.json 文件，可以使用 nova::policy 这个类，在这个类的代码中，实际是通过调用 `openstacklib::policy::base` 这个资源来完成对 policy.json 文件的配置：

```
  $policy_defaults = {
    'file_path' => $policy_path,
    'require'   => Anchor['nova::config::begin'],
    'notify'    => Anchor['nova::config::end'],
  }

  create_resources('openstacklib::policy::base', $policies, $policy
```

上面的代码使用 `create_resource` 函数动态的创建 `openstacklib::policy::base` 资源，实现配置 policy.json 文件的目的。

Puppet::Provider::Openstack::Auth

`Puppet::Provider::Openstack::Auth` 是 puppet-openstacklib 提供一个类，它被其他 openstack 模块所使用，它的主要功能是为 CLI 接口提供认证的功能，认证信息的获取方式为：

1. 从环境变量中获取认证信息
2. 如果没有从环境变量中获取到密码信息，那么读取 /root/openrc 文件

当其他 openstack 模块中的自定义资源需要使用 CLI 接口时，都会将此类中的方法加载到新定义的类中，通过这个类提供的方法完成认证。

openstack_config

openstack 中大部分项目的配置文件都使用 `ini` 格式的配置文件，puppet-openstack 社区为每个服务提供了自定义资源，用来完成对这些配置文件的管理，例如 `nova.conf` 中需要在 `[DEFAULT]` 这个 section 中添加一条配置如下：

```
[DEFAULT]
memcached_servers = 10.10.0.1:11211
```

可以使用 `nova_config` 这个自定义资源完成：

```
nova_config { 'DEFAULT/memcached_servers':
  value => '10.10.0.1:11211',
}
```

所有这些自定义资源的实现都会使用 puppet-openstack 中的基础类 `openstack_config`，因此其他 openstack 模块都会依赖此模块。

os_service_default

`nova.conf` 中需要在 `[DEFAULT]` 这个 section 中添加一条配置如下：

```
[DEFAULT]
memcached_servers = 10.10.0.1:11211
```

可以使用 `nova_config` 这个自定义资源完成：

```
nova_config { 'DEFAULT/memcached_servers':
  value => '10.10.0.1:11211',
}
```

那么，如果我想从配置文件中删除这条配置，需要这样做：

```
nova_config { 'DEFAULT/memcached_servers':
  ensure => absent,
}
```

可以看到两种代码传递的参数不同，因此我们在 openstack 模块中往往看到这种风格的代码：

```
if $memcached_servers {
  nova_config { 'DEFAULT/memcached_servers': value  => join($memcac
} else {
  nova_config { 'DEFAULT/memcached_servers': ensure => absent }
}
```

由于配置一个参数和删除一个参数需要对自定义资源传递的参数不同，在 puppet 代码中往往需要很多这种条件判断语句。为了解决这个问题，puppet-openstack 社区推行了一种新的配置方式，对 ini 格式的自定义资源进行了一些修改，例如对于 `nova_config` 这个资源来说，使用一个特定的 value 参数来表示将此资源从配置文件中删除：

```
nova_config { 'DEFAULT/memcached_servers':
  value => '<SERVICE DEFAULT>',
}
```

当给 value 参数传递 `'<SERVICE DEFAULT>'` 这个特定的字符串时，将会从 nova 的配置中删除对应的配置，这样服务就会使用默认的配置项。这个 `'<SERVICE DEFAULT>'` 字符串也是由 puppet-openstacklib 这个模块通过自定义 facter 提供的，其他模块只需要引用这个 facter 就可以了，不需要知道这个特定的字符串具体是什么。这个自定义的 facter 是使用 ruby 原生的方式，和键值对的方式定义的，这个 facter 就是 **$os_service_default**：

```
require 'puppet/util/package'

if Puppet::Util::Package.versioncmp(Facter.value(:facterversion),
  Facter.add('os_service_default') do
    setcode do
      '<SERVICE DEFAULT>'
    end
  end
end
```

有了这个自定义的 facter，其他模块中，就可以使用这个 facter 作为配置文件参数的默认值，即默认从配置文件中删除对应的选项，由服务自己来使用默认值，这样在 puppet 中也不用额外维护一套各服务配置的默认配置了，同时也减少了很多条件判断语句的代码。

小结

puppet-openstacklib 这个模块的主要功能是作为其他模块的基础模块使用，对其他模块提供自定义的资源和 facter，用于配置数据库，policy.json，以及 CLI 接口的认证，几乎所有的 openstack 模块都会使用此模块。

puppet-openstack-integration

`Puppet-openstack-integration` 模块确保社区可以持续地测试和验证使用Puppet modules部署的Openstack集群。

> 建议在阅读其他module前，优先阅读本节内容。

本节作者：余兴超

阅读级别：必读 阅读时间 **1**小时

先睹为快

如果你想要使用puppet modules部署一套all-in-one的openstack集群，那么可以在虚拟机(Ubuntu 14.04或者CentOS 7.x)的终端下执行以下命令：

```
git clone git://git.openstack.org/openstack/puppet-openstack-integr
cd puppet-openstack-integration
./all-in-one.sh
```

或者

```
curl -sL http://git.openstack.org/cgit/openstack/puppet-openstack-i
```

整个过程约需要20分钟。

我们分析以下这是怎么做到的？

all-in-one.sh是一个逻辑比较简单的脚本，其调用了run_tests.sh脚本。 这个脚本的主要作用有3点：

- 安装Puppet相关软件包，
- 执行puppet apply命令，完成相应服务的安装配置
- 安装配置tempest并相应运行smoke测试

这里面主要讲解一下是如何实现服务的安装配置，主要使用的是run_puppet函数。

```
# 函数定义
PUPPET_ARGS="${PUPPET_ARGS} --detailed-exitcodes --color=false --te

PUPPET_FULL_PATH=$(which puppet)

function run_puppet() {
    local manifest=$1
    $SUDO $PUPPET_FULL_PATH apply $PUPPET_ARGS fixtures/${manifest]
    local res=$?

    return $res
}
```

在下面连续调用了两次run_puppet函数：

```
# SCENARIO即要运行的manifests文件，决定了安装哪些服务
print_header "Running Puppet Scenario: ${SCENARIO} (1st time)"
run_puppet $SCENARIO
RESULT=$?
set -e
if [ $RESULT -ne 2 ]; then
    print_header 'SELinux Alerts (1st time)'
    catch_selinux_alerts
    exit 1
fi

# Run puppet a second time and assert nothing changes.
set +e
print_header "Running Puppet Scenario: ${SCENARIO} (2nd time)"
run_puppet $SCENARIO
RESULT=$?
set -e
if [ $RESULT -ne 0 ]; then
    print_header 'SELinux Alerts (2nd time)'
    catch_selinux_alerts
    exit 1
fi
```

核心代码讲解

目前Openstack Intra一共使用了三个测试场景，用于跑puppetopenstack的集成测试: scenario001, scenario002，scenario003.

而scenario-aio manifest是提供给想要了解和学习PuppetOpenstack项目的用户。它们之间的区别参见下表：

-	scenario001	scenario002	scenario003	scenario-aio
ssl	yes	yes	yes	no
ipv6	centos7	centos7	centos7	no
keystone	X	X	X	X
tokens	uuid	uuid	fernet	uuid
glance	rbd	swift	file	file
nova	rbd	X	X	X
neutron	ovs	ovs	linuxbridge	ovs
cinder	rbd	iscsi		iscsi
ceilometer	X			
aodh	X			
gnocchi	rbd			
heat			X	
swift		X		
sahara			X	
trove			X	
horizon			X	X
ironic		X		
zaqar		X		
ceph	X			
mongodb		X		

scenario-aio

这里我们以scenario-aio来解释它是如何部署起一个Openstack All-in-One的环境的。 scenario-aio的文件路径为: `fixtures/scenario-aio.pp`

```
#从类的名称我们就可以知道aio安装了mq，mysql，keystone，glance，neutron等服
include ::openstack_integration
include ::openstack_integration::rabbitmq
include ::openstack_integration::mysql
include ::openstack_integration::keystone
include ::openstack_integration::glance
include ::openstack_integration::neutron
include ::openstack_integration::nova
include ::openstack_integration::cinder
include ::openstack_integration::horizon
include ::openstack_integration::provision
# aio中还配置了tempest，除了默认支持的nova，keystone，glance等服务外，开启
class { '::openstack_integration::tempest':
  horizon => true,
  cinder  => true,
}
```

那么接下来，我们进入到这些被调用的类中一探究竟。为了节省篇幅，我们分别挑选了mq和glance进行解释和说明。

class openstack_integration::rabbitmq

我们可以理解为在openstack_integration的manifests目录下，所有和服务相关的类都是转发层，即对某个服务模块的调用。

在openstack_integration::rabbitmq中，通过调用class rabbitmq完成了对rabbitmq的安装和配置，并创建了一个路径为'/'的vhost，更多对rabbitmq类的介绍，请参见puppet-rabbitmq模块。

```
class openstack_integration::rabbitmq {

  include ::openstack_integration::params
  include ::openstack_integration::config

  if $::openstack_integration::config::ssl {
    file { '/etc/rabbitmq/ssl/private':
      ensure                  => directory,
```

```
      owner                   => 'root',
      mode                    => '0755',
      selinux_ignore_defaults => true,
      before                  => File["/etc/rabbitmq/ssl/private/$
    }
    openstack_integration::ssl_key { 'rabbitmq':
      key_path => "/etc/rabbitmq/ssl/private/${::fqdn}.pem",
      require  => File['/etc/rabbitmq/ssl/private'],
      notify   => Service['rabbitmq-server'],
    }
    class { '::rabbitmq':
      package_provider      => $::package_provider,
      delete_guest_user     => true,
      ssl                   => true,
      ssl_only              => true,
      ssl_cacert            => $::openstack_integration::params::ca
      ssl_cert              => $::openstack_integration::params::ce
      ssl_key               => "/etc/rabbitmq/ssl/private/${::fqdn]
      environment_variables => $::openstack_integration::config::ra
    }
  } else {
    class { '::rabbitmq':
      package_provider      => $::package_provider,
      delete_guest_user     => true,
      environment_variables => $::openstack_integration::config::ra
    }
  }
  rabbitmq_vhost { '/':
    provider => 'rabbitmqctl',
    require  => Class['::rabbitmq'],
  }

}
```

class openstack_integration::glance

挑选glance的原因在于其代码相比其他服务更简洁一些，读者理解起来会稍微容易一些。 我们可以看到其

- 调用glance::api和glance::resgistry完成了glance服务的配置
- 调用glance::notify::rabbitmq完成了MQ的配置
- 调用glance::db::mysql完成数据库的配置
- 调用glance::client完成client的配置
- 调用glance::keystone::auth完成glance keystone相关的配置
- 通过传递的参数值，选择调用glance::backend::file/glance::backend::rbd/glance::backend::swift完成后端存储的配置

```
class openstack_integration::glance (
  $backend = 'file',
) {

  include ::openstack_integration::config
  include ::openstack_integration::params

  if $::openstack_integration::config::ssl {
    openstack_integration::ssl_key { 'glance':
      notify => [Service['glance-api'], Service['glance-registry']]
    }
    Package<| tag == 'glance-package' |> -> File['/etc/glance/ssl']
    $key_file  = "/etc/glance/ssl/private/${::fqdn}.pem"
    $crt_file = $::openstack_integration::params::cert_path
    Exec['update-ca-certificates'] ~> Service['glance-api']
    Exec['update-ca-certificates'] ~> Service['glance-registry']
  } else {
    $key_file = undef
    $crt_file  = undef
  }

  rabbitmq_user { 'glance':
    admin    => true,
    password => 'an_even_bigger_secret',
    provider => 'rabbitmqctl',
    require  => Class['::rabbitmq'],
  }
  rabbitmq_user_permissions { 'glance@/':
    configure_permission => '.*',
    write_permission     => '.*',
```

```
    read_permission      => '.*',
    provider             => 'rabbitmqctl',
    require              => Class['::rabbitmq'],
  }
  class { '::glance::db::mysql':
    password => 'glance',
  }
  include ::glance
  include ::glance::client
  class { '::glance::keystone::auth':
    public_url   => "${::openstack_integration::config::base_url}:9
    internal_url => "${::openstack_integration::config::base_url}:9
    admin_url    => "${::openstack_integration::config::base_url}:9
    password     => 'a_big_secret',
  }
  case $backend {
    'file': {
      include ::glance::backend::file
      $backend_store = ['file']
    }
    'rbd': {
      class { '::glance::backend::rbd':
        rbd_store_user => 'openstack',
        rbd_store_pool => 'glance',
      }
      $backend_store = ['rbd']
      # make sure ceph pool exists before running Glance API
      Exec['create-glance'] -> Service['glance-api']
    }
    'swift': {
      Service<| tag == 'swift-service' |> -> Service['glance-api']
      $backend_store = ['swift']
      class { '::glance::backend::swift':
        swift_store_user                    => 'services:glance',
        swift_store_key                     => 'a_big_secret',
        swift_store_create_container_on_put => 'True',
        swift_store_auth_address            => "${::openstack_integ
        swift_store_auth_version            => '3',
      }
    }
```

```
      default: {
        fail("Unsupported backend (${backend})")
      }
    }
    $http_store = ['http']
     $glance_stores = concat($http_store, $backend_store)
    class { '::glance::api':
      debug                         => true,
      database_connection           => 'mysql+pymysql://glance:glance@127
      keystone_password             => 'a_big_secret',
      workers                       => 2,
      stores                        => $glance_stores,
      default_store                 => $backend,
      bind_host                     => $::openstack_integration::config::
      auth_uri                      => $::openstack_integration::config::
      identity_uri                  => $::openstack_integration::config::
      registry_client_protocol      => $::openstack_integration::config::
      registry_client_cert_file => $crt_file,
      registry_client_key_file      => $key_file,
      registry_host                 => $::openstack_integration::config::
      cert_file                     => $crt_file,
      key_file                      => $key_file,
    }
    class { '::glance::registry':
      debug                  => true,
      database_connection => 'mysql+pymysql://glance:glance@127.0.0.1
      keystone_password      => 'a_big_secret',
      bind_host              => $::openstack_integration::config::host,
      workers                => 2,
      auth_uri               => $::openstack_integration::config::keysto
      identity_uri           => $::openstack_integration::config::keysto
      cert_file              => $crt_file,
      key_file               => $key_file,
    }
    class { '::glance::notify::rabbitmq':
      rabbit_userid          => 'glance',
      rabbit_password        => 'an_even_bigger_secret',
      rabbit_host            => $::openstack_integration::config::ip_for
      rabbit_port            => $::openstack_integration::config::rabbit
      notification_driver => 'messagingv2',
```

```
    rabbit_use_ssl       => $::openstack_integration::config::ssl,
  }
}
```

小结

puppet-openstack_integration模块为PuppetOpenstack项目提供了集成测试的功能，同时也为用户提供了快速部署AIO测试环境的脚本。如果你是刚开始了解该项目，那么这个模块是快速熟悉Openstack各个基础模块的一条路径。

动手练习

1. 在aio场景中添加sahara服务
2. 在install_module.sh中的r10k命令的作用是？
3. bundler的作用是什么？

puppet-openstack-specs

1. 模块讲解

本节作者：余兴超
阅读级别：选读
阅读时间: **0.5**小时

本章节也是选读章节，仅对模块做了一个简单介绍。

`puppet-openstack-specs`模块是用于管理Blueprint design document，对Openstack了解的同学应该知道，在早年，社区使用Launchpad来管理BluePrint(BP)，后来过渡到使用这种Markdown语法编写+Code review的方式来管理功能开发文档，我觉得这是一个非常好的设计，在文档的格式定义，归档，搜索上有非常有益的尝试。

模块讲解

与PuppetOpenstack相关的specs放在specs/目录下，并根据release版本不同，划分出了不同的目录。

例如以Newton的某个BP为例：**Configuration File Deprecation Support**

它使用如下目录对BP进行详细地描述:

- Problem description #问题描述
- Proposed change #提出的改进计划
    - Alternatives #其他替代方案
    - Data model impact #对数据模型的影响
    - Module API impact #对模块API的影响
    - End user impact #对终端用户的影响
    - Performance Impact #对性能的影响
    - Deployer impact #对部署人员的影响
    - Developer impact #对开发人员的影响
- Implementation # 实现相关
    - Assignee #指派人
    - Work Items #任务列表

- Dependencies #依赖
- Testing #测试
- Documentation Impact #对文档的影响
- References #参考链接

总结为四个字：非常专业。

- puppet-openstack-specs
    - 模块讲解

puppet-openstack-cookiebutter

本篇是选读章节，和Openstack部署没有直接的管理，本节推荐从事开发并维护自研Puppet module的读者。

1.先睹为快

`puppet-cookiebutter` 模块用于快速生成一个符合PuppetOpenstack代码风格的新module。 不想看下面大段的代码说明，已经跃跃欲试了？

Ok，我们开始吧！

打开虚拟机终端并输入以下命令：

```
# 请使用pip安装cookiecutter
$ cookiecutter puppet-openstack-cookiecutter/
project_name [YOURPROJECTNAME without 'puppet-']: test
version [0.0.1]:
year [2016]:
```

接着进入到puppet-test模块查看其目录结构，包含manifests/，spec/和lib/目录，manifests目录下创建了通用代码目录，例如db/下的mysql.pp, posygresql.pp, sync.pp等。 同时puppet-test添加了LICENSE和README等文件，在此基础上可以开始进行新module的开发工作：

```
|-- LICENSE
|-- README.md
|-- lib
|   `-- puppet
|       |-- provider
|       |   `-- test_config
```

```
|       |       `-- ini_setting.rb
|       `-- type
|           `-- test_config.rb
|-- manifests
|   |-- config.pp
|   |-- db
|   |   |-- mysql.pp
|   |   |-- postgresql.pp
|   |   `-- sync.pp
|   |-- db.pp
|   |-- init.pp
|   |-- keystone
|   |   `-- auth.pp
|   |-- logging.pp
|   |-- params.pp
|   `-- policy.pp
|-- metadata.json
|-- spec
|   |-- classes
|   |   |-- test_db_mysql_spec.rb
|   |   |-- test_db_postgresql_spec.rb
|   |   |-- test_db_spec.rb
|   |   |-- test_keystone_auth_spec.rb
|   |   |-- test_logging_spec.rb
|   |   `-- test_policy_spec.rb
|   |-- shared_examples.rb
|   `-- unit
|       |-- provider
|       |   `-- test_config
|       |       `-- ini_setting_spec.rb
|       `-- type
|           `-- test_config_spec.rb
`-- tests
    `-- init.pp
```

模块讲解

在介绍 `puppet-openstack-cookiebutter` 模块之前，先了解一下 `Cookiecutter` ：

> 一个用于创建项目模板的命令行工具集，最初的目的是用于创建Python项目。

使用该工具可以非常快速地生成一个Python软件包项目：

```
cookiecutter https://github.com/audreyr/cookiecutter-pypackage.git
```

关于cookiecutter的使用就不再展开，更详细的文档说明请参见：

- Documentation: https://cookiecutter.readthedocs.io
- GitHub: https://github.com/audreyr/cookiecutter
- PyPI: https://pypi.python.org/pypi/cookiecutter

本节不涉及任何代码的讲解，仅简要介绍一下其工作原理：cookiecutter使用了Python的Jinja2库对预置模板进行渲染，在puppet-openstack-cookiebutter模块中有一个目录是 `puppet-{{cookiecutter.project_name}}` ，其目录结构如下：

```
.
|-- lib
|   `-- puppet
|       |-- provider
|       |   `-- {{cookiecutter.project_name}}_config
|       `-- type
|-- manifests
|   |-- db
|   `-- keystone
|-- spec
|   |-- classes
|   `-- unit
|       |-- provider
|       |   `-- {{cookiecutter.project_name}}_config
|       `-- type
`-- tests
```

可以看到模板变量 `{{cookiecutter.project_name}}` ，那么在何处定义呢？

打开 `cookiecutter.json` 文件就可以看到：

```
{
    "project_name": "YOURPROJECTNAME without 'puppet-'",
    "version": "0.0.1",
    "year":"2016"
}
```

3.小结

`puppet-openstack-cookiebutter` 是一个辅助性模块，通过它可以快速地创建一个Openstack服务模块的所有基础代码目录和文件。如果在公司内部恰巧有开发内部模块的需求，那么通过它可以快速地构建出一个新模块。

4.动手练习

1. 阅读contrib/bootstrap.sh脚本并解释其使用用途
2. 在puppet-中添加一个新文件guide.md，并生成一个新模块puppet-cook

puppet-modulesync-configs

1. 先睹为快
2. 模块讲解

本节作者：余兴超

阅读级别：选读

阅读时间: **0.5**小时

本节为选读章节，推荐有兴趣的读者阅读。

先睹为快

我们通过 `puppet-openstack-cookiebutter` 模块的contrib/bootstrap.sh脚本来说明：

```
# Step 4: Retrieve the puppet-modulesync-configs directory and conf
#
git clone https://review.openstack.org/openstack/puppet-modulesync-
pushd puppet-modulesync-configs/
cat > managed_modules.yml <<EOF
---
  - puppet-$proj
EOF
cat > modulesync.yml <<EOF
---
namespace:
git_base: file://$tmp_var/cookiecutter/
branch: initial_commit
EOF

# Step 5: Run msync and amend the initial commit
#
msync update --noop
pushd modules/puppet-$proj
md5password=`ruby -e "require 'digest/md5'; puts 'md5' + Digest::MD
```

```
sed -i "s|md5c530c33636c58ae83ca933f39319273e|${md5password}|g" spe
git remote add gerrit ssh://$user@review.openstack.org:29418/openst
git add --all && git commit --amend -am "puppet-${proj}: Initial co

This is the initial commit for puppet-${proj}.
It has been automatically generated using cookiecutter[1] and msync

[1] https://github.com/openstack/puppet-openstack-cookiecutter
[2] https://github.com/openstack/puppet-modulesync-configs
"


echo "

-----------------------------------------------------------------
The new project has been successfully set up.

To submit the initial review please go to ${tmp_var}/puppet-modules
and run git review.

Happy Hacking !
"
```

通过以上步骤，可以为一个新puppet module同步Gemfile,Rakeflie等文件，其中执行同步操作的关键命令是:

```
# 注意需要使用gem安装msync命令
msync update
```

模块讲解

打一个比方，modulesync-config模块类似于Openstack的requirements项目。用于管理Gem包的依赖，Rakefile的配置等等。它有几个重要的文件：

- `config_defaults.yml`：第一层级的键表示模块被管理的文件名
- `.sync.yml`：出现在各自module中，将覆盖 `config_defaults` 中的值
- `managed_modules.yml`：被管理的module列表
- `modulesync.yml`：传递到Modulesync命令行的键值对参数对

关于 `puppet-modulesync-configs` 的用途比较单一，因此不再深入展开，更详细的解释参见其模块说明。

- puppet-modulesync-configs
    - 先睹为快
    - 模块讲解

puppet-openstack_spec_helper

This will be done in `v0.2`

- puppet-openstack_spec_helper

puppet-stdlib

本节作者：余兴超
阅读级别：必读
阅读时间: **1.5**小时

先睹为快

`puppet-stdlib` 是由Puppet官方提供的标准库模块。这是一个聚宝盆，几乎在前面介绍的Openstack模块中都会使用到它。因为DSL作为一个不完整的语言（不是男人），缺少某些内置魔法和特性会让程序员们抓狂。
例如，在Python中借助内置库可以轻松地做数值比较：

```
max(1,2,3)
```

那么在原生Puppet中，你只能望而兴叹。因此我们需要——puppet-stdlib模块！

```
# 和Python不同的是,max函数须在语句中使用。
$largest=max(1,2,3)
notify {"$largest":}
```

核心资源讲解

在这个模块中，它提供了以下Puppet资源：

- Stages
- Facts
- Functions
- Defined resource types

- Types
- Providers

接下来，我们将挑选一些使用频率较高的资源进行讲解。

Run Stages

我们知道为了保证resources间的执行顺序，可以使用 `require` , `subscribe` , `notify` 等元参数或者使用链式标记来指定resources间的执行顺序。例如：

```
package {'ntp':
  ensure => present
}
# ntp.conf的配置依赖于ntp软件包的安装
file {'/etc/ntp.conf':
  ensure  => present,
  require => Package['ntp']
}
# ntpd进程的运行依赖于ntp软件包和配置文件
service {'ntpd':
  ensure    => running,
  subscribe => Package['ntp'],File['/etc/ntp.conf']
}
```

但是在 `class` 和 `class` 之间就没法使用这些方法去标记类之间的执行顺序了。那么 `Run stages` 允许将指定分组的类按照不同的stage来顺序执行。

`main` stage

在Puppet中，默认只有一个stage（ `main` ）。所有的资源都被默认自动地关联到这个stage上，如果你不显式地为Resources指定stage，那么所有的资源都会在`main stage·阶段允许。

使用定制stage

使用定制stage和其他资源的调用方式完全相同，除了有一点硬性要求是：

> Each additional stage must have an order relationship with another stage

例如，我们可以使用以下方式进行声明：

```
# 通过元参数的方式
stage { 'first':
  before => Stage['main'],
}
# 通过链式箭头的方式
stage { 'last': }
Stage['main'] -> Stage['last']
```

接下来，我们只需要将stage关联到class：

```
  # stage作为元参数出现
class { 'ntp':
  stage => first,
}
```

使用stdlib::stages

终于讲到了正题了：`stdlib::stages` 类声明了各种run stages用于基础设施，语言运行时和应用的部署。它提供了以下stages：

- setup
- main
- runtime
- setup_infra
- deploy_infra
- setup_app
- deploy_app
- deploy

使用起来也很简单，不用先声明，直接使用即可：以下为代码示例：

```
  node default {
    include stdlib
    class { java: stage => 'runtime' }
  }
```

`file_line` type

配置文件的管理是CMS中最主要的目标之一。对于 `INI` 格式的配置文件的管理方式有多种不同的配置方式。但是对于一些非格式化的配置文件来说，其配置管理通常都是选择使用 `template` 的方式进行管理。

`file_line` type的出现，使得我们有了一种更轻量的方式去管理非格式化配置文件。它的实现与正则匹配和替换类似。

我们来看看实际的使用吧：

```
# 添加指定行
# 在/etc/sudoers文件中确保`%sudo ALL=(ALL) ALL`被正确添加
file_line { 'sudo_rule':
  path => '/etc/sudoers',
  line => '%sudo ALL=(ALL) ALL',
}
```

在实际使用中，除了添加之外，更多的场景是替换：

```
# 修改指定行
# match参数允许使用正则表达式来精确匹配文本行中的内容
file_line { 'bashrc_proxy':
  ensure => present,
  path   => '/etc/bashrc',
  line   => 'export HTTP_PROXY=http://squid.puppetlabs.vm:3128',
  match  => '^export\ HTTP_PROXY\=',
}
```

在一些场景下，我们需要删除配置文件中指定的行：

```
#删除指定行
#match_for_absence参数决定在ensure => absent下，是否执行操作
#mutiple 确保在多行匹配时继续执行操作（否则报错）
file_line { 'bashrc_proxy':
  ensure            => absent,
  path              => '/etc/bashrc',
  line              => 'export HTTP_PROXY=http://squid.puppetlabs.v
  match             => '^export\ HTTP_PROXY\=',
  match_for_absence => true,
  multiple          => true
}
```

ensure_packages

`ensure_packages` 接受array/hash类型的软件包列表，并确保它们被正确地安装。其实和 `package` 资源的使用是相似的，但最大的不同点，在于 `ensure_packages` 函数可以被安全地多次定义，而不会发生duplicated resource的错误。 下面举例说明其使用：

```
# array类型，其中'ksh'被重复传了2次，但可以安全通过编译
ensure_packages(['ksh','openssl'], {'ensure' => 'present'})
ensure_packages(['ksh','vim'], {'ensure' => 'present'})
#
ensure_packages({'ksh' => { enure => '20120801-1' } ,
                 'mypackage' =>
                         { source => '/tmp/myrpm-1.0.0.x86_64.rpm
                {'ensure' => 'present'})
```

`ensure_resource` 与其类似，这里就不再展开说明。

validate_xxx

stdlib中提供大量的validate_xxx前缀开头的函数，它们的作用是对输入源进行验证，常被用于变量类型检查中，有点类似于assert的断言，若为false，则退出catalog的编译。

例如: `validate_bool` 验证所有传入的参数是否都为true/false布尔类型。

```
$iamtrue = true
validate_bool(true)
validate_bool(true, true, false, $iamtrue)
```

`validate_hash` 函数用于验证传入的参数是否为hash类型

```
$my_hash = { 'one' => 'two' }
validate_hash($my_hash)
```

is_xxx

stdlib中也提供了另一类型的验证函数，仅当验证通过后，会返回true/false的布尔值。

`is_array` 验证传入参数是否为array类型:

```
$numbers=[1,2,3]
is_array($numbers)
```

`is_ipv4_address` 验证传入参数是否为ipv4类型的地址:

```
$my_ip='10.0.0.88'
is_ipv4_address($my_ip)
```

其他函数

在 `puppet-stdlib` 模块中还有一些用于处理string，hash，array等类型的函数，我们会在Openstack模块和基础模块中去单独介绍。

小结

`puppet-stdlib` 模块的出现极大地增强了puppet对于各种数据类型的处理能力，提供诸多功能，请读者详细阅读 `puppet-stdlib` 的 `readme.markdown` 文件，或者在线阅读stdlib的文档。

动手练习

1. 验证服务器是否具有eth1，如果有则验证传入的$ipaddress_eth1是否为ipv4类型地址。
2. 将['china','japan','korea']转化为['china-travel','japan-travel','koera-travel']
3. 判断一台服务器是否为虚拟机，如果是则将libvirt_type设置为'qemu',否则设置为'kvm'
4. 从$bigclass namespace中查询变量$var的值，其中$bigclass='foo::bar' (提示: `getvar`)

puppet-openstack_extras

This will be done in `v0.2`

本节作者：余兴超

阅读级别：必读

阅读时间: **1**小时

`puppet-openstack_extras` 模块是Openstack模块的工具集，它主要提供以下功能：

- composition classes
- HA utilities
- monitoring functionality

它替代了早年的 `[puppet-openstack](https://github.com/stackforge/puppet-openstack)` 模块（已弃用）。

先睹为快

- puppet-openstack_extras
    - 先睹为快

Openstack自动化部署最佳实践

> Fools ignore complexity. Pragmatists suffer it. Some can avoid it. Geniuses remove it.
>
> — Alan J. Perlis, “Epigrams in Programming”

我们在一一介绍了众多基础模块，Openstack模块以及公共库模块后，是否就可以抡起袖子开始做大规模线上部署了?
答案是No。

哎，有话好好说，你们别丢臭鸡蛋啊...
读者：%#@($#@!，你骗我读了那么长的文档，竟然不能用于线上部署！

部署是一项复杂的系统工程，前期的架构设计，硬件规划和兼容性和性能测试，采购，上架，裸机操作系统安装，网络配置，这些都完成了，才到了软件部署阶段。熟悉本书介绍的Puppet modules，加上对于Puppet的基础使用，的确是可以胜任Openstack集群的部署和配置管理了。 但是我们想把积累了5年的Openstack部署经验和4年的Puppet管理线上集群各种经验和教训归纳成最佳实践告诉给读者，以避免不合理的设计，为后期的运维管理埋下不稳定因素。

本章内容

我们将其归纳为以下几大块。

1.代码管理相关：

> 代码的规范与否体现了一个程序员的素质，代码的管理则反映了一家公司对待技术的态度。

那么代码规范体现在以下几点：

- 使用版本控制工具进行管理
- 符合一门语言的通用代码风格要求
- 完整的文档，包含commit消息，代码注释，架构文档等
- 不使用花式技巧

2.配置管理相关:

- Hiera
- Node Definition
- Environment
- PuppetDB

3.任务编排相关:

- ClusterShell
- Ansible

4.运维原则相关:

- 配置管理操作的基本守则

如何做好模块管理

不同于其他的Openstack项目，puppet modules是一个数量庞大的存在。以我们当前在使用中的puppet modules为例，就已经多达96个（破百计日可待）。

依赖管理

目前有三种管理依赖的方式：

- Git submodules 通过git submodule的方式管理各个子模块
- Puppet module tool 可以使用puppet forge基于module名称和版本来搜索和安装module
- Librarian-puppet ruby bundler的扩展，使用Puppetfile来管理

我们分别就这三种方式依次介绍一下，我们这里不说哪种方法最好，但我们会说明我们根据什么原因最终选择了哪种方法。

1.Puppet module tool

该方法使用metadata.json文件来管理每个module之间的依赖关系，以puppet-nova为例:

```
"dependencies": [
    { "name": "puppetlabs/apache", "version_requirement": ">=1.0.0
    { "name": "duritong/sysctl", "version_requirement": ">=0.0.1 <
    { "name": "openstack/cinder", "version_requirement": ">=8.0.0 <
    { "name": "openstack/glance", "version_requirement": ">=8.0.0 <
    { "name": "puppetlabs/inifile", "version_requirement": ">=1.0.0
    { "name": "openstack/keystone", "version_requirement": ">=8.0.0
    { "name": "puppetlabs/rabbitmq", "version_requirement": ">=2.0.
    { "name": "puppetlabs/stdlib", "version_requirement": ">=4.0.0
    { "name": "openstack/openstacklib", "version_requirement": ">=8
    { "name": "openstack/oslo", "version_requirement": "<9.0.0" }
  ]
```

2.Librarian-puppet

librarian-puppet支持从Modulefile或者metadata.json读取依赖，或者使用独立的Puppetfile。例如，社区的puppet-openstack_integration项目里就包含了Puppetfile:

```
## OpenStack modules
mod 'aodh',
  :git => 'https://git.openstack.org/openstack/puppet-aodh',
  :ref => 'master'

mod 'barbican',
  :git => 'https://git.openstack.org/openstack/puppet-barbican',
  :ref => 'master'
  ...
```

可以使用以下命令安装其所依赖的module:

```
  librarian-puppet install --verbose
```

3.git submodule

git submodule可以同时管理多个独立的项目，同时保持提交的独立。这也是目前我们所选择的方式。 我们根据Puppet Module的类型将其划分成了三个项目（你可以理解为modules的group）：

- sunfire 内部自研服务模块
- storm Openstack服务相关模块
- karma 运维系统相关模块

我们会为storm创建多个分支，例如:liberty,mitaka。在dev和test环境会使用git命令来切换代码，而在生产环境则会使用RPM包的方式来管理。这样做的好处是：

- 遵循线上代码统一使用软件包管理的方式
- dev和test环境可以随时修复代码并且灵活切换

Hiera

1.简介和历史

Hiera是基于键值查询的数据配置工具，Hiera是一个可选工具，它的目标是：Hiera makes Puppet better by keeping site-specific data out of your manifests

它的出现使得代码逻辑和数据可以分离管理。

在Puppet 2.x版本时代，Hiera作为一个独立的组件出现，若要使用则需要单独安装。在3.x版本之后，Hiera被整合到Puppet的代码中去。Hiera是Hierarchal单词的缩写，表明了其层次化数据格式的特点。

1.1 实例说明

在使用hiera前，一个常见的manifests文件是这么编写的：

```
node "kermit.example.com" {
  class { "ntp":
    servers    => [ '0.us.pool.ntp.org iburst','1.us.pool.ntp.org i
    autoupdate => false,
    restrict   => [],
    enable     => true,
  }
}

node "grover.example.com" {
  class { "ntp":
    servers    => [ 'kermit.example.com','0.us.pool.ntp.org iburst
    autoupdate => true,
    restrict   => [],
    enable     => true,
  }
}

node "snuffie.example.com", "bigbird.example.com", "hooper.example.
  class { "ntp":
    servers    => [ 'grover.example.com', 'kermit.example.com'],
    autoupdate => true,
    enable     => true,
  }
}
```

在使用了Hiera之后，manifests文件发生了如下变化：

```
node "kermit.example.com", "grover.example.com", "snuffie.example.c
  include ntp
  # or:
  # class { "ntp": }
}
```

所有的数据设置移到了hiera中:

```
---
ntp::restrict:
 -
ntp::autoupdate: false
ntp::enable: true
ntp::servers:
  - 0.us.pool.ntp.org iburst
  - 1.us.pool.ntp.org iburst
  - 2.us.pool.ntp.org iburst
  - 3.us.pool.ntp.org iburst
```

2.hiera.yaml配置文件

hiera.yaml是Hiera唯一的配置文件，它其中只有少数几个配置参数，但决定了Hiera不同的使用方式。

2.1 文件路径

在puppet.conf中通过设置hiera_config参数来设置hiera.yaml文件的路径，默认值为：$confdir/hiera.yaml

> 注意Puppet 4.x以上时，默认值变更为$codedir/hiera.yaml

2.2 参数详解

以下为hiera.yaml配置文件的默认值:

```
---
:backends: yaml
:yaml:
  # on *nix:
  :datadir: "/etc/puppetlabs/code/environments/%{environment}/hiera
  # on Windows:
  :datadir: "C:\ProgramData\PuppetLabs\code\environments\%{environm
:hierarchy:
  - "nodes/%{::trusted.certname}"
  - "common"
:logger: console
:merge_behavior: native
:deep_merge_options: {}
```

参数名称	类型	说明
:hierarchy	str或array	每一行表示静态或动态的数据源，动态源是指使用了%{variable}格式的变量，hiera采用从上往下的顺序读取数据源。
:backends	str或array	yaml或json，默认值:yaml
:logger	str	决定warning和debug级别日志的发送位置: console (messages送到STDERR), puppet (messages送到Puppet日志系统), noop (messages静默) Puppet会将值覆盖为puppet，无论设置为其他任何一个值。
:merge_behavior	str	native (default) — 仅合并最顶层key; deep — 递归合并; 在遇到冲突的key时, 低优先级会被使用。deeper — 递归合并; 在遇到冲突的key时, 高优先级会被使用。
:deep_merge_options	array	当:merge_behavior设置为deep或deeper时使用
:yaml / :json :datadir	str	数据源文件的查找路径

Automatic Parameter Lookup

Hiera 是用来存储数据的地方，那么当Puppet 代码中需要从hiera 中读取某个数据时，我们可以在代码中使用hiera() 函数的方式，在hieradata 中查找某个键值存储的数据，例如在hieradata 文件test.yaml 中定义了：

```
foo: bar
```

那么，我可以在Puppet 代码中获取foo 这个键对应的值：

```
$text = hiera('foo')
notify { "$text": }
```

foo 这个键对应的值通过hiera 函数获取到之后被保存在了$text 变量中。

除了使用hiera(), hiera_include(), hiera_array(), hiera_hash() 等函数去hieradata 中读取之外，Puppet 还会自动从Hiera 中查找类参数，查找键为myclass:parameter_one（即类名::参数名）。

在定义一个Puppet 类时，可以定义默认的参数值，例如下面的myclass 类的参数$parameter_one 使用了默认值"default text"：

```
class myclass ($parameter_one = "default text") {
  file {'/tmp/foo':
    ensure  => file,
    content => $parameter_one,
  }
}
```

当我们调用myclass 这个类时，Puppet 遵循如下方式来设定$parameter_one 这个参数的值：

1. 如果在调用这个类的时候，显式的向其传递了参数值，那么Puppet 使用显式传递的值作为参数的值。
2. 如果调用类时没有传递参数的值，那么Puppet 会自动从Hiera 中查询参数的值，查找时使用:: 做为查找的键（例如上面的myclass 类的prarameter_one 参数，查找键为myclass::parameter_one）
3. 如果方法1 和2 都没有获取到值，那么Puppet 会使用类定义中参数的默认值作为参数的值（例如myclass 中prameter_one 参数的默认值为"default text"）
4. 如果1 至3 都没有获取到值，那么Puppet 将会直接报错，代码的编译将被中

断。

上面的方法2 是Puppet 最有趣的地方，因为Puppet 会自动从Hiera 中查找参数的值，我们可以在代码中使用include 语句来调用一个类，不需要对其传递任何参数值，所有的参数传递都可以将参数值写到Hiera 中，Puppet 会自动从Hiera 中读取类的参数。例如，我想调用上面定义的myclass 类，并且$parameter_one 的参数值为"ustack"，参数的传递使用Hiera 来完成。那么我需要在Hiera 中写入下面的值：

```
myclass::parameter_one: 'ustack'
```

在代码中，调用myclass 类：

```
include myclass
```

这里不用对myclass 传递参数，myclass 会自动读取Hiera 中对parameter_one 定义的值，即$parameter_one 的值在调用时为'ustack'

参考文档

http://docs.puppetlabs.com/hiera/latest/

提交规范

> 注意： 本节主要以内部的最佳实践分享，非强制要求。

Puppet Modules使用git做版本控制工具，因此我们这里不会去重复git文档中的基础知识点。

我们这里主要提及的有以下几点：

- 每次commit中应该包含哪些代码？
- commit message的格式应该怎么写？
- 代码提交是否应该和其他系统关联？

1.提交规范说明

- 每次提交只包含相关的Puppet代码逻辑和单元测试
- 第一行是commit的简短描述
- 在第一行和后面段落之间插入一个空行
- 提供针对本次commit的详细描述（可选）
- 本次提交的类型,Type是:(BF|NF|RF|OT|BugFix|NewFeature|ReFactor|Other)
- 关联JIRA issue，Jira: link链接
- 每行不能超过72个字符

1.1 Type类型说明

Type类型	全称	说明
BF	BugFix	漏洞，问题修复
NF	NewFeature	新特性开发
RF	ReFactor	代码重构，架构重构，文档补充等
OT	OTher	其他类型，例如，添加.gitreview，添加.gitignore，添加mailmap等与项目无关操作

1.2 Label类型说明

Label类型	全称	说明
Type	Commit-Type	必填，本次提交的类型
Jira	Jira-Link	必填，Jira的链接
FC	Forward Compatibility	可选，是否向前兼容，原则上不允许不向前兼容的代码
CT	CriTicality	可选，危险程度，一般只适用于线上变更项目

1.3 格式样例

```
commit 7c027d40e2b616ba57f7c69f8162a6311461a566
Author: [removed]
Date:   Fri Aug 28 10:14:28 2015 -0700
    Ensure setuptools is somewhat recent

    Due to bugs in older setuptools version parsing
    we need to set a relatively new version of setuptools
    so that parsing works better (and/or correctly).

    This seems especially important on 2.6 which due to
    a busted setuptools (and associated pkg_resources) seems
    to be matching against incorrect versions.

    Type: BF
    Jira: DEVOPS-453
    Change-Id: Ib859c7df955edef0f38c5673bd21a4767c781e4a
```

正确使用环境

Environment这个概念是比较容易理解的，你可以联想到开发环境，测试环境，线上环境等等。 是的，environment的目的就是为了将不同类型的host分组。 我们都知道module的默认路径是放在/etc/puppet/modules。 假设我现在要开发一个puppet-apache模块，和线上环境使用的puppet-apache模块代码不一样，但是/etc/puppet/modules不是只能放一个模块吗？ 因此，每个environment支持独立的Puppet modules和main manifest（节点定义文件）。

假如没有environemnt，而同时有生产，开发和测试环境，每套环境的Puppet代码都不尽相同，那么就需要搭建3台用于不同环境的Puppet Master。用Puppet Environment可以很容易的解决这个问题，使用Puppet Environment可以建立dev, production, test 三套环境，并且每套环境的代码都可以不同。这样就可以用一台Puppet Master管理多个集群环境。

Directory Environments vs. Config File Environments

使用过2.x或者更早版本的同学，应该了解或使用过Config file environemtns。目前，Puppet支持两种定义环境的方式：

- Directory Environments
- Config File Environments

> 注意，Config File Env这种需要修改配置文件的方式已经被历史的潮流抛弃，所以我们只会介绍directory environment。

启用Directory Environment

在Puppet Master上启用Directory Environment，需要在puppet.conf中定义以下参数：

> environmentpath = /etc/puppet/environments

此参数定义了一个目录，此目录下的每一个子目录都是一个environment。

Environment目录结构

上面说过，使用directory environment 的方式，每一个目录都是一个环境，这个目录可以包含环境自身的配置，模块和节点定义。directory environment（下面会使用环境目录替代）遵循下面的规律：

- 环境目录的名称即环境名称（知道为啥config file environment必须死了吧。）
- 环境目录必须放在environmentpath下，默认是在/etc/puppet/environements下
- 应该包含一个modules目录，属于该环境默认的module路径
- 应该包含一个manifests目录，属于该环境默认的节点定义路径
- 可以包含一个environment.conf文件，用于自定义当前环境modulepath和manifest设置来指定此环境的模块查找路径和节点定义路径。，比如说test和dev两个环境可以共用一个manifest目录。

在我们的线上业务中，一般使用了如下几个环境：

- dev
- test
- pre_production
- production

在Agent 端指定Environment

在Puppet master 上定义了多套环境之后，在agent 段需要指定本机使用的环境，否则就会使用默认的production 环境。在puppet.conf 中定义environment 参数来指定agent 所属的环境，例如指定agent 为liberty 环境：

> [agent]environment = liberty

- 正确使用环境
  - Directory Environments vs. Config File Environments
  - 启用Directory Environment
  - Environment目录结构
  - 在Agent 端指定Environment

转发层规范

转发层（composition layer）模块实际上是对第二部分所介绍的模块的调用。 官方社区曾经有一个转发层模块称为puppet-openstack，在Juno版本时被标记为弃用。为什么社区不推荐转发层模块呢？因为这玩意和自家业务结合得非常紧密。比如，Fuel有自己的转发层模块cluster,ctask有自己的转发层sunfire。

那么关于它的最佳实践是什么？

2016年1月份，我们刚完成了对转发层sunfire的重构，甩掉了许多的历史包袱。关于转发层，我们有以下几点原则需要严格遵守:

逻辑清晰

概括来讲就是在转发层中，不允许出现对resource的直接调用，所有转发层的类抑或定义，只能直接调用基础模块，Openstack模块中的类或定义。

打个不是非常恰当的比方:

```
class sunfire::api(){
  # 这块代码就应该被移除，使用include ::nova::client来替换
  package {'python-novaclient':
    ensure => present,
  }
  include ::nova
  include ::nova::api
}
```

数据和逻辑分离

我们在早期使用Puppet时并没有使用到Hiera，因此转发层承载了大量参数的默认值设置。这样做的好处是，我们需要使用到的参数都赋有一个合理默认值，但是坏处是数据和逻辑没有完全分离开。 比如说，我想查询一下目前线上集群$keystone_user_password的值，可能是在转发层的代码中，也可能是记录在hieradata中。另外一个不好的地方就是代码会变得非常冗余。

打个比方:

```
class sunfire::api(
  $nova_db_password = 'nova',     #先定义一个参数
){
 class {'nova::db::mysql':
   db_password => $nova_db_password, #把该参数值传给真正需要赋值的参数
 }
}
```

完全分离的写法:

```
class sunfire::api(){
 include ::nova::db::mysql
}
```

角色松耦合

我们习惯把转发层中的每个class称之为角色，这样比较形象，例如:

- sunfire::api 表示API节点
- sunfire::mq 表示MQ节点
- sunfire::loadbalancer::l7 表示7层负载均衡

那么API角色中，又包含了nova/cinder/neutron/glance/... api,keystone等大量的服务。同时我们又要满足某些情况下，某些服务不启用的需求。例如，某用户表示，他们不需要使用neutron server而启用nova-network。 有两种方式来满足这种要求：

- 在sunfire::api添加开关:

```
class sunfire::api(
  $enable_neutron = true,
){
  if $enable_neutron {
    include ::neutron::server
  }
}
```

- 在main manifests文件中声明：

```
'xxx api node' {
include ::neutron::server
}
```

- 转发层规范
    - 逻辑清晰
    - 数据和逻辑分离
    - 角色松耦合

代码风格

每种语言都会有自己特定的风格，每个人也会会有自己的风格，那么在协作的项目中，我们就需要去遵循大家都认可的规范。

熟悉语法和风格

在提交代码到代码审查系统前，请在本地开发环境使用 `Puppet-lint` 工具检查一遍。

如何标记弃用(Deprecation)

所有**patch**必须保持向后兼容（**backward compatible**）

这意味着：

- 不能破坏原有接口（参数弃用至少保持一个周期，并要添加warning信息)
- 不能改变参数的默认值(除非有一个好理由，并在commit消息里解释清楚原因）

代码一致性

目前社区维护了大量的modules，我们需要保持代码的一致性，这体现在使用统一风格的代码和注释，每个独立参数选项保持唯一。例如，在添加新参数前，请查找是否在其他类或者模块中已经实现，如果没有，则保持和其他参数相同风格的方式添加。

空值的参数

当你需要设置一个空的(nil)参数时，使用 `undef` 值。不要使用''或者false。

文档

- 确保所有的参数都添加了正确的文档，lint会检查这些文档是否被添加。

- 尽可能地保持examples.pp文件更新
- 必须情况下，为你的代码添加注释(temporary workarounds, TODO等)

Standalone vs C/S 模式

Standalone(单机模式)

Puppet可以很容易地以单机模式下执行配置管理的工作。在这种模式下Puppet将会在本地 编译并且执行catalog，无需和Puppet Server通讯，适合一些简单的配置管理任务或者Puppetserver节点的自部署工作。

`puppet` 命令支持多种使用方式，可以在终端下输入:

```
puppet -v apply test.pp  # 最常用的使用方式
puppet apply -e 'include ::ntp::server'  #执行一段puppet代码
puppet apply --catalog catalog.json #指定执行一个catalog文件
```

Client/Server(客户端/服务器端模式)

C/S模式是大家都熟悉的运行模式。在这种模式下，Puppet agent端被部署在了被管理的服务器上，Puppet master(server)端部署在管理服务器上。

TODO：C/S流程图

如何选择？

脱离业务场景去谈孰优孰劣都是不合时宜的。

适合单机模式的场景

如果是个人开发环境的配置管理，少数服务器的配置管理，以及业务逻辑比较简单的情况下，推荐使用单机模式，简单方便。

适合C/S模式的场景

在涉及正式线上环境的情况下，我们推荐是C/S模式。优势非常明显：

1. 安全管理和权限控制

配置管理代码中其实包含了大量的敏感信息，其一旦发生泄漏或者权限被越界，就发导致严重的安全问题。

在单机模式下，每台服务器都会拿到完整的puppet代码和hiera数据，试想一台Apache服务器上放着MySQL服务器的配置管理代码和管理员用户名密码是何其危险的事情！

另外，在CS模式下，agent端和server端通过SSL进行通信，并且可以根据节点的FQDN做细粒度的控制，试想一台伪造自己是数据库服务器的节点，在向服务器端请求就轻而易举地拿到了数据库节点的catalog也是非常危险的事情。

2.支持高级特性

在C/S模式下，通过开启storeconfigs参数，配合PuppetDB，可以使用Puppet中诸如exported_resources等高级特性。

3.集中式管理

在单机模式下，若要批量执行puppet配置管理任务，一般选择使用集群管理工具例如clustershell等做批量的变更操作。 在C/S模式下，agent既可以定期地从server端获取最新的catalog，也可以由Server端主动地发送更新指令。 此外，在C/S模式下，agent端在完成配置应用的任务后，可以发送report给Server端或者其他服务器。

Puppet版本的选择

TODO: Version pic

Puppet 2.x

目前Puppet 2.x的最新版本是2.7.x。除非是遗留系统保留大量的第三方模块，无法升级到其他版本，否则我们极力推荐读者从2.x升级到3.x版本。

Puppet 3.x

Puppet 3.x的最新版本是3.8.7。相比2.x，其catalog的编译速度提升了50%，并且也包含了4.x里的新增特性(通过额外配置开启）。如不是特殊原因，我们推荐读者从3.x升级到4.x版本。

Puppet 4.x

在2016年9月22日，Puppet已发布了4.7.0版本，之前我们曾建议读者谨慎使用4.x，但时至今日，我们推荐直接使用4.x版本。

目前PuppetOpenstack社区已经放弃对Puppet3的支持(https://review.openstack.org/#/c/383739/)。

因此，我们的建议是：

- 如果您正在使用Puppet3，请谨慎升级到Puppet4。
- 如果你正在计划使用Puppet，直接使用Puppet4。

我们会在下一节去详细介绍Puppet 4.x的显著变化。

Puppet4新特性和变化

激动人心的改进

Puppet4的第一个正式版本于2015年4月15日发布，截止到2016年9月22日，Puppet已正式发布了4.7.0版本。

Puppet4与3.x版本相比有两点不同：很多的变化，很大的变化。毫不夸张地说Puppet4是一个全新的项目！

速度，速度，还是速度

Puppet4使用函数式编程语言Clojure对Puppet Master进行了重写，Puppetlabs公司并为此新建了一个项目:puppetserver。此外，PuppetDB也使用Clojure进行了重写。

如此脱胎换骨的变化，最主要的目的是为了提升性能，官方给出的数据是:

> 相比Puppet3，Puppet4有2~3倍的性能提升。

这是一个非常吸引人的提升！要知道从Puppet2到Puppet3所带来约50%的性能提升，就让我们感动不已了！

在以往的实际生产中，我们遇到过多次来自于master端的性能瓶颈，在一个数千台规模有近百个Openstack集群规模的环境中，我们使用了多台物理+虚拟服务器来作为puppet master节点，管理着大量的服务，一旦遇到高并发的编排任务时，master端的CPU几乎处于100%的状态，超时时间设置为120秒的情况下，仍然会出现不少由于编译catalog超时而导致agent报错的情况。即使我们通过改进代码，水平扩展，组件拆分，参数调优，更换硬件等多种组合办法，但是受Puppet本身的语言性能瓶颈，对于Puppetmaster的性能我们并不满意。而Puppet4从根本上改进了性能问题。

PuppetDB也是主要瓶颈之一，像resource export,virtual resource等高级特性，以及facts,catalog的缓存都会使用到PuppetDB，虽然这些高级特性很炫酷而且也很实用，但是非常非常消耗资源。这使得我们在过去非常地谨慎甚至刻意去削减像Puppet高级特性的使用，这也是PuppetOpenstack社区禁止提交含有这些高级特性的代码的原因之一（另一个原因是有些高级特性无法再单机模式下使用）。

稳定性和鲁棒性的提升

此外，Puppet4一开始就拥有面向服务的架构：

- 由于Clojure语言的天生优势，拥有良好的并发和互斥控制能力，而且可以使用

丰富的Java Library，是作为后端服务开发的理想选择。

- Puppetlabs公司开发了一个Clojure框架Trapperkeeper framework：为了支撑长期运行的应用和服务而生，从而保证Puppet服务的稳定性和鲁棒性。

全新的Parser

- 新的Parser支持lambdas和iteraion！再也不用使用tricky的creates_resources函数了：

```
$a = [1,2,3] each($a) |$value| { notice $value }
```

- 全新的parser还直接支持数据类型检查，再也不用stdlib里的validate_string等函数了:

```
class ntp (
    Boolean $service_manage = true,
    Boolean $autoupdate     = false,
    String  $package_ensure = 'present',
    # ...
) {
   # ...
}
```

- 另外一个亮点是直接支持插值式函数调用:

```
notice "This is a random number: ${fqdn_rand(30)}
```

- 支持链式赋值，代码可以变得更简洁了：

```
$a = $b = 10
```

除了以上几点，还有其他诸多特性，受篇幅限制，不再一一举例。

“不变”的agent

目前，puppet-agent仍然使用Ruby来维护。不过JVM可以支持Ruby的Java版本：JRuby。因此在未来，puppet-agent不排除可能会从JRuby过渡到Clojure。

不兼容的改动

Puppet4既然做了重写，因此有大量与Puppet3不兼容的变化。这些细节对于Puppet3用户来说是最关心的地方。

包管理方式的变化

过去，我们需要在服务器上单独安装Puppet,Facter,Hiera,Mcollective等多个组件才能获得相应的功能和特性。

在Puppet4中，安装Puppet不再需要安装多个软件包，而是采用AIO(All-in-One)的方式来简化软件包的管理，例如 `puppet-agent` 中包含以下组件：

- Facter 3.4.x
- CFacter 0.4
- Hiera 1.3.x
- Mcollective 2.9.x
- Ruby 2.1.5
- OpenSSL 1.0.0r

Puppetlabs将这种AIO的包管理方式称之为Puppet Collections(PC)，每个PC其实对应着一个软件仓库(repo)，为用户提供了Facter/Ruby/Puppet等组件的匹配矩阵。 下表给出了PC中主要软件包中整合的组件。

软件包名	包含组件
`puppet-agent`	Puppet, Facter, Hiera, MCollective, pxp-agent, root certificates, Ruby, Augeas
`puppetserver`	Puppet Server，依赖 `puppet-agent`
`puppetdb`	PuppetDB
`puppetdb-termini`	PuppetServer与PuppetDB交互的Plugin

要在服务器上启用新版本的Puppet4，只需要执行一行简单的命令：

- 在基于RPM的系统下使用以下命令：

```
yum localinstall http://yum.puppetlabs.com/puppetlabs-release-p
```

- 在基于Deb的系统下使用以下命令：

```
# curl -O http://apt.puppetlabs.com/puppetlabs-release-pc1-whee
```

通过这种集中式的软件仓库管理方式，用户可以移除过去puppetlabs-release中的production，dependencies，devel等多个仓库。

注意 `puppet-agent` 不会自动升级老版本的 `puppet` 软件包(建议使用deb或rpm来管理软件包的升级)

配置文件/目录的路径变化

1. 软件包的安装目录变更为 `/opt/puppetlabs`
2. 可执行文件已移动到 `/opt/puppetlabs/bin`
3. `confdir` 从 `/etc/puppet/` 变为 `/etc/puppetlabs/puppet`
4. `ssldir` 从 `$vardir/ssl` 变为 `$confdir/ssl`
5. puppetserver的配置文件放置在 `/etc/puppetlabs/puppetserver`
6. mcollective的配置文件放置在 `/etc/puppetlabs/mcollective`
7. 所有的module/manifest/data从 `confdir` 移到 `codedir`
    - `codedir` 默认路径是 `/etc/puppetlabs/code`
    - 包含 `environments` 目录
    - 包含全局的 `modules` 目录（可选）
    - 包含hiera.yaml配置文件
    - 包含 `hieradata` 目录

其他路径变化

- `puppet agent` 的 `vardir` 已经移动到 `/opt/puppetlabs/puppet/cache`
- `rundir` 已经移动到 `/var/run/puppetlabs`

`Directory Environment` 正式启用

过去多年的Config File Environment将被正式移除。默认的environmentpath是 `$codedir/environments` 。

以新建一个 `production` 环境为例：

- 将modules放置到 `$codedir/environments/production/modules`
- 将main manifest放置到 `$codedir/environments/production/manifests`

你仍然可以使用 `$codedir/modules` 作为全局modules，并用 `default_manifest` 设置来配置一个全局的 `main manifest` 。

不再使用Ruby1.8.7

由于使用了AIO的包管理方式，Puppet不再使用系统自带的Ruby解释器，将直接使用Ruby 2.1.5版本。

下一代Puppet语言的改动

重点来了，Puppet4最重要的变化是重写了parser和evaluator，在Puppet 3.x中可以通过在puppet配置文件中开启 `Future Parser` 来使用，在Puppet4中该parser已经成为"present parser"，那么过去的parser正式退出舞台。

新parser包含了迭代，变量类型检查等诸多新特性。并且，新parser对于数值，空字符串和'udenf/nil'比较提供更好的检查机制。

除了核心模块的变动以外，还有一些炫酷的特性。

- 在PuppetMaster加载新的Puppet代码不再需要重启server服务
- EPP(Embeded Puppet)将支持直接使用Puppet来编写inline和基于文件模，不再需要使用ERB，避免用户在Puppet和Ruby之间来回切换。
- 支持使用Puppet来编写functions。

`Puppet Kick` 等将被移除

所有的项目在历史发展过程中，都会有很多的妥协和不良设计，Puppet项目从2到3很多旧有的特性只是被标记为废弃，并没有从代码库中移除，借助Puppet4版本的重构，大约60000行"technical debt"类型的代码被移除。 较为熟知的有以下:

- `puppet kick` 命令

- `inventory` 服务
- `couchDB` facts terminus
- `ActiveRecord` stored config
- puppet.conf中 `master` section

HTTP API的变化

Puppet4中的另一个重要变化是master和agent通讯的URLs发生了变化。因此Puppet3的agent将无法和Puppet4的server端通信。例如：

- 在Puppet3中url是"http://localhost:8140/production/node/foo"
- 在Puppet4中url变成了"http://localhost:8140/puppet/v3/node/foo?environment=production"。

`puppet doc` 和 `tagmail` 被移除

由于 `puppet doc` 命令依赖RDoc，而RDoc与最新版本的ruby不兼容，因此在Puppet4代码中被移除，如果要继续使用，可以通过puppetlabs-strings模块来提供类似的功能。

同理， `tagmail` 被移除，可以通过puppetlabs-tagmail模块来找到它。

Resource Type/Providers的变化

这里举几个重要的变化：

- 在Puppet3中，若用户没有设置allow_virtual属性，会有废弃的警告信息，在Puppet4中该警告会被移除，allow_vritual默认会从false变为true。

内部API和实现的变化

这些变化只会影响到Puppet内部ruby方法和库的调用接口，对终端用户的使用没有任何影响。

被废弃的特性

Rack和WEBrick Web服务器被废弃

Rack和WEBrick Web服务器过去常用于开发和简单验证，目前已在Puppet 4.1中标记为弃用，计划在5.0中移除。

主要配置参数

Puppet4有多达200个配置参数，不过用户需要关心的参数大约为30个。这里我们只是简单介绍 `puppet.conf` 中的主要参数。

Agent端

基础参数

- `server` : Puppet Master的地址，默认值是 `puppet`
    - `ca_server` : Puppet CA的地址，仅在多master模式使用
    - `report_server` : Puppet report server的地址，仅在多master模式使用
- `certname` ：node的证书名称，默认使用FQDN
- `environment` ：agent向master端请求的environment。默认是 `prodcution` 。

运行相关

- `noop` : agent仅在模拟运行并输出运行结果
- `nice` : 指定agent运行的nice值，防止agent在应用catalog时占用过多的CPU资源
- `report` : 是否发生report，默认为true。
- `tags` ： 限制Puppet只运行含有指定tags的resources。
- `trace` , `profile` , `graph` , `show_diff` ：用于debug agent运行结果
- `usecacheonfailure` : 在master端无法返回一个正确的catalog时，是否回退执行上一个正确的catalog。默认是true，如果是开发环境，建议修改为false。
- `prerun_command` 和 `postrun_command` ：在Puppet执行前后运行的命令，若返回值非0，则Puppet执行失败。

服务相关

- `runinterval` : Puppet的运行间隔
- `waitforcert` : Puppet请求证书签名的频率。当agent端第一次启动时，agent会提交一个CSR(certificate signing request)到ca server，该证书可能是自动签名(autosign)，或者需要人工批准，而这段时间无法预估，因此需要设置一个时间段，默认是2m。
- `splay` 和 `splaylimit` ：为每次Agent的定时执行添加一个随机数时间，用于避免惊群效应的发生。
- `daemonize` :是否以进程方式运行，配合cron使用时，应设置为false。
- `onetime` : 是否执行完成后退出，配合cron使用时，应设置为true。

Server端

多数参数对于单机模式运行的Puppet同样适用。在CS模式下，这些参数应该放置在[master]下；在单机模式下，这些参数应该放置在[main]下。

主要参数

- `dns_alt_names` : Puppet Master可以使用的DNS主机名列表(alt表示a list)。agent用到的 `server` 参数值必须和此参数或者server端的 `certname` 匹配。
    - 注：该参数仅适用于初始化生成Puppet master证书阶段。
- `environment_timeout` : master从environment加载数据的缓存时长。设置为0，禁用缓存，为了更好的性能，可以将其设置为 `unlimited` ，直到下次重启master才会重新加载environment配置。

- `enviromentpath` : environment的查找路径，默认值: `$codedir/environments`
- `basemodulepath` : 所有环境的模块路径，会被所有的环境使用，默认值是: `$codedir/modules:/opt/puppetlabs/puppet/modules`
- `reports` : 选择处理report的hander,默认值是 `store` 。

Server其他配置

pupept server除了 `puppet.conf` 之外，还有拥有其他的配置文件，其默认的配置文件路径是: `/etc/puppetlabs/puppetserver/conf.d` 。这些配置文件使用HOCON格式，可以在保留JSON语义格式的前提下，提高可读性。在conf.d目录下包含以下配置文件:

- global.conf
- webserver.conf
- web-routes.conf
- puppetserver.conf
- auth.conf
- master.conf (deprecated)
- ca.conf (deprecated)

例如，常见的几个参数配置有以下：

- `puppet-admin` ：授权可以访问admin接口的client
- `jruby-puppet` : 调优JRuby时提供更多细节信息
- `JAVA_ARGS` : 设置Puppet Server的内存分配。

总结和建议

相比Puppet2到Puppet3的版本升级，Puppet4不仅是纯粹的新功能和性能提升，更像是对Puppet的全新重构，摒弃了过去留下来的历史负担和诟病。但是，Puppet4所带来的不兼容性，导致对于Puppet3用户，尤其是Puppet 3.3版本之前的用户，若想要把线上的业务代码升级到Puppet4，需要花费不少的精力。

分享我们的升级经验：约在16年初，我们将线上的Puppet版本从3.3升级到了3.7，并提前使用了future parser特性（Puppet4中的parser）。16年9月，我们使用了半周时间完成了对Puppet4升级调研并给出调研报告，在Jira上列出计划和任务分工，实际使用了一周时间完成了96 Puppet Module代码更新和测试，并在生产环境上线。

因此我们的建议是：即使Puppet4令人激动人心，但配置管理系统的升级一定要谨慎对待，提前做好计划和回滚方案，充分测试，分步骤操作。

参考文档

- https://docs.puppet.com/puppet/4.0/reference/whered_it_go.html
- https://docs.puppet.com/puppet/4.0/reference/release_notes.html
- https://puppet.com/blog/welcome-to-puppet-collections
- http://www.infoworld.com/article/2687553/devops/puppet-server-drops-ruby-for-clojure.html

- https://docs.puppet.com/puppet/latest/reference/puppet_collections.html

Puppet的能与不能

Openstack云平台是一个复杂的软件栈，涉及到大量的配置，服务，软件包等等多种系统资源的管理，人工管理的方式必然带来最终不可维护，人工失误等诸多问题。因此，我们需要使用一套统一框架解决配置管理上的问题。我们在过去发现了一点就是，工程师们通常喜欢在一个系统/工具上去套所有的应用场景。

首先，我们要明白任何一个工具/语言/系统都不是万能的。我们在使用一个工具/语言/系统前，必须深刻理解它的能力和局限。我们既然选择了Puppet作为配置管理系统，就应知道它能做什么，不能做什么。

什么场景下选择使用Puppet?

Puppet适合用于以下场景：

- network,host,dns等文件的配置管理
- ssh,ntp,nscd等服务的状态管理
- MySQL,Apache,RabbitMQ等软件的包管理
- Openstack软件的包安装，配置文件管理以及服务状态的管理
- Puppet原生resource type，如ssh,host等
- Puppet第三方模块中的扩展resource，如keystone_user,mysql_database等

什么场景下不选择使用Puppet?

Puppet不适合使用以下场景：

- 源码文件管理

  有人可能会使用 `file` resource来管理一些项目的脚本，比如zabbix的plugin scripts，这些代码文件通常作为静态文件放置在files/目录下，在agent应用catalog的阶段，从puppetserver下载到各个服务器上。这样做有好多缺点：

  - 首先, 每次代码文件的更新必须要更新相应模块，业务代码和部署代码完全耦合。
  - 其次，所有线上业务没统一的代码管理方式，在我们内部，所有项目必须使用RPM包的方式进行统一管理

  - 最后，每次执行Puppet Agent都会对每个文件单独计算hash值，并在服务器端做hash值比较，且会把旧文件备份到备份文件目录下，此操作会消耗客观的CPU和IO资源。

- 软件包的依赖管理

  例如在计算节点上，nova-compute依赖bridge-utils包，有些工程师喜欢在nova::compute里去添加一个 `package` 资源来确保在计算节点上安装此包。正确的做法是在nova的spec文件里，对openstack-nova-compute组件新增一条包依赖关系。 我们应该明确包的依赖管理应该交给软件包管理工具去做。

- 二进制文件的管理

  我们可能会为一个业务系统添加一下方便管理，查询，统计或者清理的脚本，一些工程师的做法是将这些二进制可执行文件也丢到了puppet module的file/目录下，随着项目的发展会出现大量的二进制文件，那么它们的归宿，要么放到该项目的tools目录，或者单独作为一个项目存在，例如openstack_tools。

- 服务的初始化操作

  Puppet中有个 `exec` 资源，有些工程师拿它来写非常复杂的bash脚本。这就类似使用Python的subprocess去写非常复杂的bash脚本一样，实现方式非常丑陋，而且低效。例如Nova服务的 `db sync` 操作，复杂的实现逻辑已经封装到了Python脚本中，Puppet只是通过exec{'nova-db-sync'}去调用 `nova-manage db sync` cli接口。

  因此在遇到业务逻辑非常复杂或者代价太大就应该交给项目去实现，对外提供操作简单的接口，然后交给Puppet去调用，而不是由Puppet去实现。

```
exec { 'nova-db-sync':
   command => "/usr/bin/nova-manage ${extra_params} db sync",
   refreshonly => true,
   logoutput => on_failure
}
```

- 服务状态的监控和恢复

  有些工程师认为可以把Puppet的runinterval改成60s，这样就可以使用Puppet做频繁的状态收敛来确保服务一直是处于运行状态，虽然在一定程度上，可以确保服务的运行状态，但这里有两个问题：

  - Puppet既不是监控系统也不是专业的服务状态管理，60s的执行间隔对于服务来说，简直是太长了
  - Server端每次编译catalog会消耗大量资源，在集群数量增长或者Puppet代码逻辑复杂度提高后，你将会发现catalog的编译时间都已经超过了60s的执行间隔。

- 角色间或节点间的依赖管理处理

  Puppet本身没有编排能力，只能处理同个节点内类之间或者服务之间的依赖关系，这是Puppet最大的硬伤。因此，Puppet公司后来就收购了Mcollective来弥补编排的短板。我们这里推荐使用Ansible来做集群编排，Ansible作为后起之秀，提供了基于YAML格式的配置管理和编排能力。

其他部署工具

在前面的章节中，我们介绍了如何使用PuppetOpenstack项目来完成Openstack的自动化部署，但是PuppetOpenstack只是部署模块，并不具备编排功能，没有GUI界面，没有CLI工具，因为我们也要关注社区deployment相关工具的发展。在本章，将介绍当前社区的主流部署工具和产品，并介绍其适用场景。

首先介绍当前产品成熟度最高的Fuel，然后是后起之秀Kolla，接着是老牌Packstack和TripleO，最后是Openstack-Ansible(OSA)，以及DevStack。

那么在最后一节，我们会介绍如何去亲手设计和定制一个满足企业或客户需求的部署工具。

Fuel

What is Fuel?

Fuel 是由 Mirantis 公司开发的一个开源的 OpenStack 部署和管理工具，也是最为流行和易用的 OpenStack 部署管理工具。Mirantis 使用 Fuel 来快速的给客户交付一套生产可用的 OpenStack。Fuel 使用了 Puppet/Cobbler/Mcollective 等开源工具，同时使用 Python/Ruby 开发了部分自有服务，Fuel 的最大特点是它能提供 Web 界面用于安装部署和管理 OpenStack，除此之外它还有如下特点：

- 硬件的自动发现
- 通过 WebUI 对硬件进行配置，如网络配置，磁盘分区配置
- 可以管理多个 OpenStack 集群
- 完善的 HA 架构支持
- 部署前的检查和网络连通性验证
- 部署后的集群健康性检查

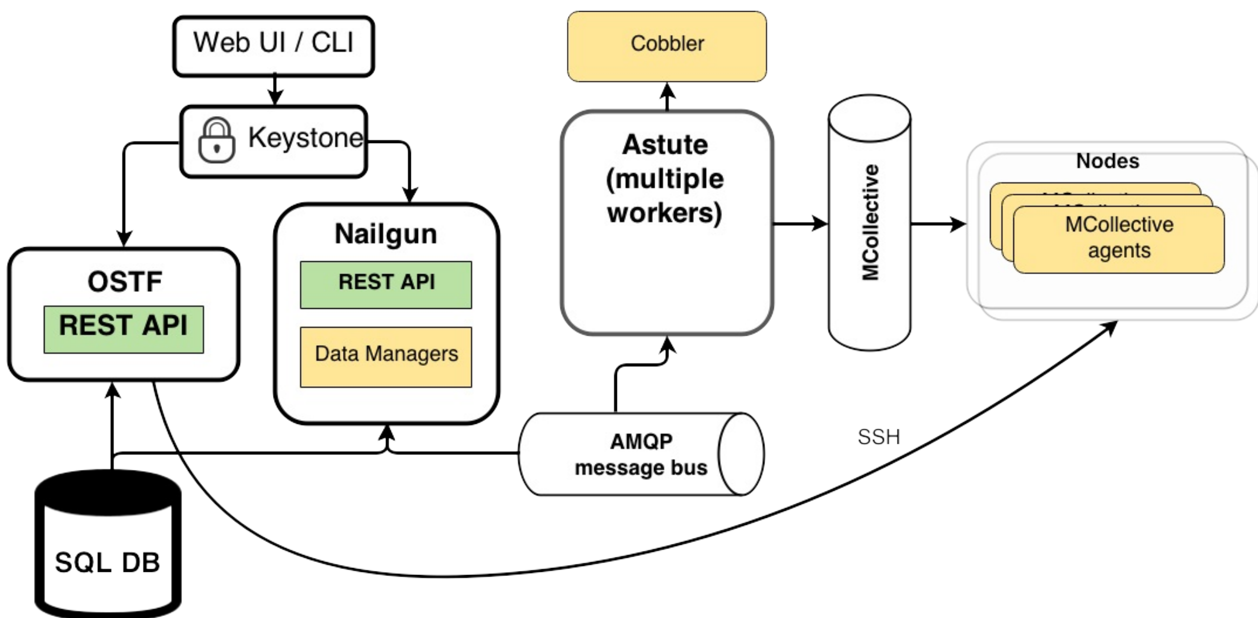
Online demo

如果想快速的体验 Fuel，可以去 http://demo.fuel-infra.org:8000/ 体验 Fuel 的 demo 版本。

Fuel 架构

Fuel 的架构包含：

- **Fuel Master Node,** 安装了 Fuel 的服务器，用于完成 Provisioning, Configuration 以及 Fuel Slave 的 PXE booting 等功能。
- **Fuel Slave Node,**Fuel Slave Node 就是用于部署控制，计算，存储服务的服务器。



Fuel 内部由众多组件组成，其中一些是 Fuel 自研发的(Nailgun, OSTF, Astute)，一些是第三方的开源组件(Puppet, Cobbler, MCollective)。

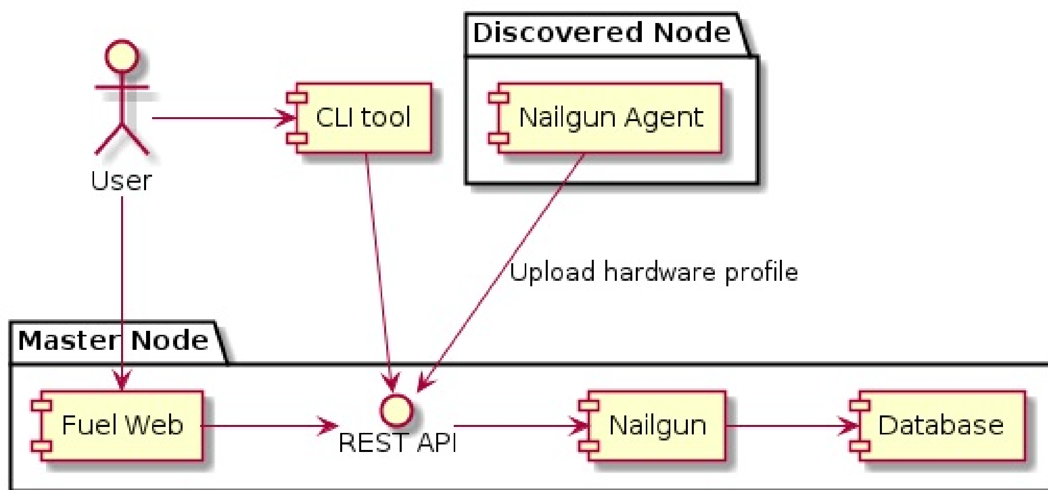
- Fuel 的 UI 是一个单页应用（JS）。
- **Nailgun** 是 Fuel 的核心组件，使用 Python 开发，它对外提供 REST API，它的主要功能是管理和保存配置数据，并处理部署的编排逻辑。它通过数据库来保存配置数据，通过 AMQP 来下发指令。
- **Astute** 相当于 Nailgun 的 worker，它接收 Nailgun 下发的指令，并完成对应的工作，它主要的工作主要是与 Cobbler/Puppet 等进行交互，使得这些服务对外提供一个抽象的异步接口。它通过 XML-RPC 来对 Cobller 进行管理和调用，通过 MCollective 来进行在节点上运行 Puppet 或者执行脚本。Astute 与 Nailgun 之间通过 AMQP 来进行数据通信。
- **Cobbler** 提供 provisioning 服务，Fuel 正在测试使用 Ironic 来替换 Cobbler（POC 阶段）
- **Puppet** 提供软件的部署服务。
- **Mcollective Agent** 用于执行脚本，运行 Puppet 等任务。
- **OSTF**(OpenStack Testing Framwork, or Health Check) 是 Fuel 的健康检查组件，它是一个独立的组件，可以脱离 Fuel 单独使用。它的主要作用是 OpenStack 的部署后功能性校验。

Fuel 架构中最核心的部分就是 Fuel Master Node。它包含了所有用于管理其他节点的所有服务，包括安装操作系统，创建 OpenStack 云环境等等。其中 Nailgun 是最重要的服务。它是一个 RESTful 的应用，包含了所有的业务逻辑。用户通过 Fuel Web 接口或者 CLI 接口来与它进行交互。例如创建新的环境，编辑配置，将节点与角色关联，并开始 OpenStack 集群的部署等等。

Nailgun 将所有的配置数据存储在 PostgreSQL 中，其中包含了节点的硬件配置，角色，环境等配置，以及当前的部署状态信息等等。

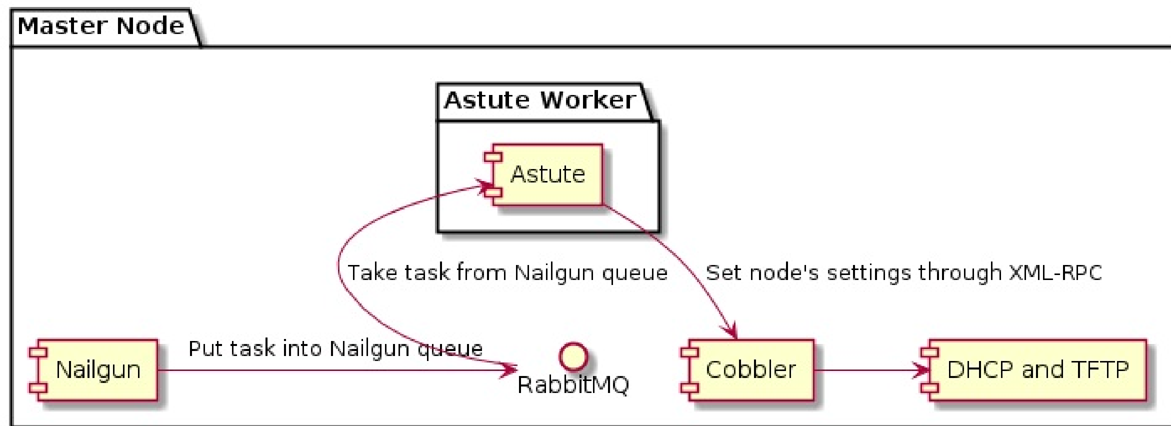
服务器发现

被管理节点启动时（PXE）会通过 master 节点上的 PXE 服务器提供的一个特殊的 bootstrap 镜像启动。这个镜像会运行一个特殊的脚本，即 Nailgun agent。nailgun-agent.rb 会收集节点的硬件信息并通过 REST API 将这些信息注册到 Nailgun 中，这样就完成了服务器的自动发现，可以在面板中给这些被发现的主机分配角色。



集群部署

当用户配置了一个新环境后，部署过程就开始了。Nailgun 服务会创建一个带有环境配置信息的 JSON 文件，并将这个文件发送到 RabbitMQ 上。这个信息应该由进行部署的进程收到，这个进程叫做 Astute。

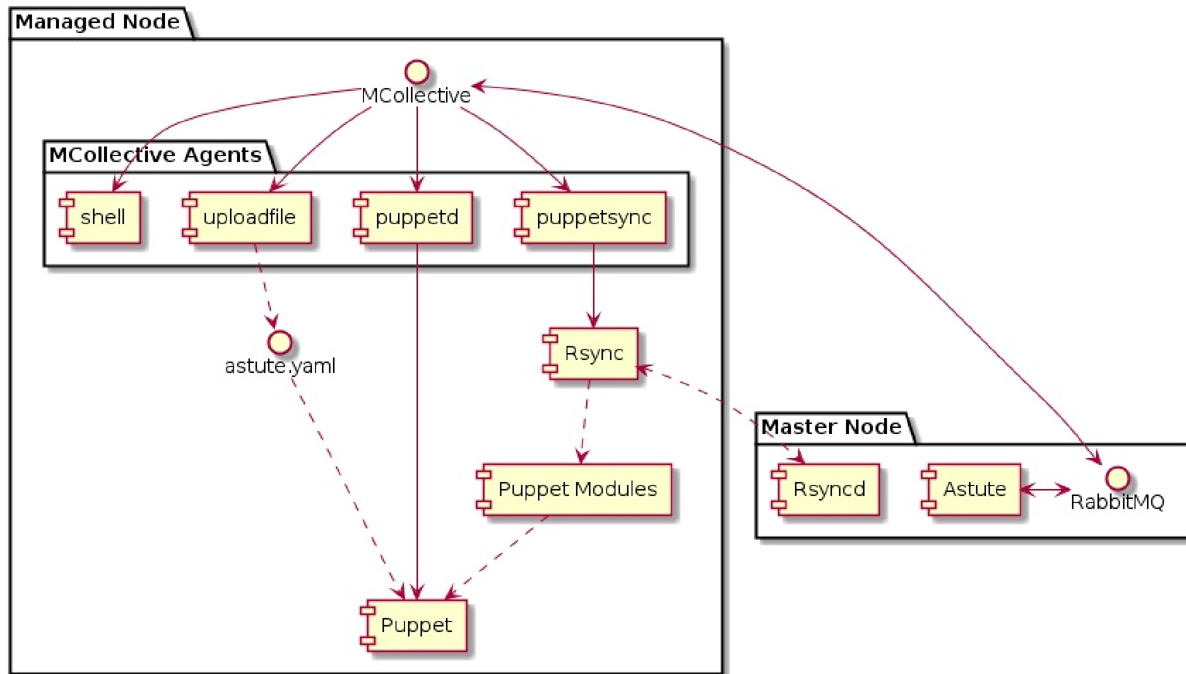


Astute 工作进程会监听 RabbitMQ 队列。它使用它的 Astute 库来完成所有的部署工作。首先，它会对环境的节点提供 provisioning 服务。Astute 使用 XML-RPC 来在 Cobbler 中设置节点的配置，并通过 MCollective agent 来重启节点来让 Cobbler 安装操作系统。Cobbler 是一个批量部署系统，它通过 DHCP 和 TFTP 服务来提供 PXE 服务。

Astute 把被管理节点需要执行的操作发送到 RabbitMQ 中。MCollective server 会在所有安装完操作系统的节点上启动并监听 MQ 中的消息，当收到消息后，会执行响应的操作并传递接收到的对应参数。MCollective agent 就是一些 Ruby 脚本，这些脚本用于完成相应的操作。

当被管理节点的操作系统安装完成后，Astute 就会开始部署 OpenStack 服务。首先，它使用 **uploadfile agent** 将节点的配置文件分发到每个节点上，文件路径为 **/etc/astute.yaml**。这个文件包含了用于部署此节点的所有的变量和配置。

接下来，Astute 使用 **puppetsync** agent 来同步所有的 Puppet 模块和 manifest 文件。同步是通过 rsync 连接到 Master 节点的 rsync server 完成的，它会下载最新版本的 Puppet 模块和 manifest 文件。



当模块同步完成后，Astute 会通过运行 Puppet 的 **site.pp** 来进行部署工作。MCollective agent 通过 **daemonize** 工具来运行 puppet，类似于这种命令：

```
deamonize puppet apply /etc/puppet/manifests/site.pp
```

Astute 会周期性的检查 agent 的执行进度，如果部署完成，它会通过 RabbitMQ 将状态报告至 Nailgun。

Puppet 会读取 **astute.yaml** 文件的内容作为 fact 变量，并将其解析到 **$fuel_settings** 这个变量中。

当 Puppet 运行结束后，Astute 获取 Puppet 运行结果的概要文件，并将结果汇报至 Nailgun。用户可以通过 Fuel Web 或者 CLI 来查看运行的进度和结果。

Fuel 还提供 **puppet-pull** 脚本，开发者可以使用这个脚本来手动从 Master 同步 manifest 并在节点上运行。

Astute 还会完成一些其他的操作，例如：

- 生成和上传 SSH key
- 使用 **net_verify.py** 来验证网络的连通性
- 部署完成后上传 CirrOS 镜像到 Glance 中
- 当有新的节点加入时，更新所有节点的 **/etc/hosts**
- 当 Ceph 节点部署后更新 RadosGW map

当一个环境被删除时，Astute 通过 Mcollective agent 来删除所有点的的启动删除，并重启节点，这些节点会通过 bootstrap 镜像重新引导，可以用于新环境的部署。

Kolla

简介

Kolla是OpenStack Big Tent Governace下的一个项目，项目的目标是

> To provide production-ready containers and deployment tools for operating OpenStack clouds.

Kolla使用Docker容器和Anisble playbooks来实现这个目标。Kolla是开箱即用的，即使你是个新手也可以很快的使用kolla快速部署你的openstack集群。Kolla也允许你根据实际的需求来定制化的部署。

kolla目前已经可以部署以下openstack项目

- Aodh
- Barbican
- Bifrost
- Ceilometer
- Cinder
- CloudKitty
- Congress
- Designate
- Glance
- Gnocchi
- Heat
- Horizon
- Ironic
- Keystone
- Kuryr
- Magnum
- Manila
- Mistral
- Murano

- Neutron
- Nova
- Rally
- Sahara
- Senlin
- Swift
- Tempest
- Trove
- Vmtp
- Watcher
- Zaqar

可以部署的基础组件包括

- Ceph 做cinder/nova/glance存储后端
- [collectd (https://collectd.org), [InfluxDB (https://influxdata.com/time-series-platform/influxdb/), and [Grafana (http://grafana.org) for performance monitoring.
- [Elasticsearch (https://www.elastic.co/de/products/elasticsearch) and [Kibana (https://www.elastic.co/de/products/kibana) 日志分析
- [HAProxy (http://www.haproxy.org/) and [Keepalived (http://www.keepalived.org/) 高可用
- [Heka (http://hekad.readthedocs.org/) 分布式可扩展的日志系统
- [MariaDB and Galera Cluster (https://mariadb.com/kb/en/mariadb/galera-cluster/) 高可用数据库
- [MongoDB (https://www.mongodb.org/) Ceilometer和Gnocchi的数据库后端
- [Open vSwitch (http://openvswitch.org/)
- [RabbitMQ (https://www.rabbitmq.com/) 消息队列

Kolla体验

可以参照kolla官方文档https://github.com/openstack/kolla/blob/master/doc/quickstart.rst 进行部署。

Kolla解决的问题

可配置的灵活架构

可以看下默认的多节点架构

```
# These initial groups are the only groups required to be modified.
# additional groups are for more control of the environment.
[control]
# These hostname must be resolvable from your deployment host
control01
control02
control03

# The above can also be specified as follows:
#control[01:03]     ansible_ssh_user=kolla

# The network nodes are where your l3-agent and loadbalancers will
# This can be the same as a host in the control group
[network]
network01

[compute]
compute01

# When compute nodes and control nodes use different interfaces,
# you can specify "api_interface" and another interfaces like below
#compute01 neutron_external_interface=eth0 api_interface=em1 storag

[storage]
storage01

[baremetal:children]
control
network
compute
storage

# You can explicitly specify which hosts run each project by updati
# groups in the sections below. Common services are grouped togethe
```

```
[kibana:children]
control

[elasticsearch:children]
control

[haproxy:children]
network

[mariadb:children]
control

[rabbitmq:children]
control

[mongodb:children]
control

[keystone:children]
control

[glance:children]
control

[nova:children]
control

[neutron:children]
network

[cinder:children]
control

[memcached:children]
control

[horizon:children]
control

[swift:children]
```

```
control

[heat:children]
control

[murano:children]
control

[ironic:children]
control

[ceph-mon:children]
control

[ceph-rgw:children]
control

[ceph-osd:children]
storage



# Additional control implemented here. These groups allow you to co
# services run on which hosts at a per-service level.
#
# Word of caution: Some services are required to run on the same ho
# function appropriately. For example, neutron-metadata-agent must 
# same host as the l3-agent and (depending on configuration) the dh

# Glance
[glance-api:children]
glance

[glance-registry:children]
glance

# Nova
[nova-api:children]
nova
```

```
[nova-conductor:children]
nova

[nova-consoleauth:children]
nova

[nova-novncproxy:children]
nova

[nova-scheduler:children]
nova

[nova-spicehtml5proxy:children]
nova

[nova-compute-ironic:children]
nova

# Neutron
[neutron-server:children]
control

[neutron-dhcp-agent:children]
neutron

[neutron-l3-agent:children]
neutron

[neutron-lbaas-agent:children]
neutron

[neutron-metadata-agent:children]
neutron
```

默认我们会把haproxy放到network节点，如果我想把haproxy放到一个单独的节点，那么我只需要到这样修改

```
-[haproxy:children]
-network
+[haproxy]
+haproxy01
+haproxy02
```

配置文件管理

每个openstack服务都运行在一个容器中，那kolla是怎么管理openstack的配置的呢？我们拿nova-compute的配置管理来举例

首先kolla会使用ansible为nova-compute生成一份配置文件放在/etc/kolla/nova-compute/目录下。

```
#nova_custom_config默认是/etc/kolla/configs/nova
#node_config_directory默认是 /etc/kolla
- name: Copying over nova.conf
  merge_configs:
    vars:
      service_name: "{{ item }}"
    sources:
      - "{{ role_path }}/templates/nova.conf.j2"
      - "{{ node_custom_config }}/global.conf"
      - "{{ node_custom_config }}/database.conf"
      - "{{ node_custom_config }}/messaging.conf"
      - "{{ node_custom_config }}/nova.conf"
      - "{{ node_custom_config }}/nova/{{ item }}.conf"
      - "{{ node_custom_config }}/nova/{{ inventory_hostname }}/nov
    dest: "{{ node_config_directory }}/{{ item }}/nova.conf"
  with_items:
    - "nova-api"
    - "nova-compute"
    - "nova-compute-ironic"
    - "nova-conductor"
    - "nova-consoleauth"
    - "nova-novncproxy"
    - "nova-scheduler"
    - "nova-spicehtml5proxy"
```

大家可能会注意到kolla使用merge_configs来完成配置文件的合并，那么merge_configs是干什么的呢?顾名思义，merge_configs就是把多个配置文件合成一个,kolla为什么要这样做呢？ openstack配置选项非常多但是真正需要管理的则很少，对这部分选项kolla使用模版的方式管理，同时由于merge_configs的使用，使得用户可以非常方便的添加自己的定制化选项。比如你部署kolla在一台虚拟机上，你必须使用QEMU hypervisor来替代KVM hypervisor。那么你可以在/etc/kolla/config/nova/nova-compute.conf中添加以下配置

```
[libvirt]
virt_type=qemu
```

merge_configs的代码在 ansible/action_plugins/merge_configs.py

启动容器时/etc/kolla以docker卷的形式挂载到/var/lib/kolla/config_files目录下

```
- name: Starting nova-libvirt container
  kolla_docker:
    action: "start_container"
    common_options: "{{ docker_common_options }}"
    image: "{{ nova_libvirt_image_full }}"
    name: "nova_libvirt"
    pid_mode: "host"
    privileged: True
    volumes:
      - "{{ node_config_directory }}/nova-libvirt/:{{ container_con
      - "/etc/localtime:/etc/localtime:ro"
      - "/lib/modules:/lib/modules:ro"
      - "/run/:/run/"
      - "/dev:/dev"
      - "/sys/fs/cgroup:/sys/fs/cgroup"
      - "kolla_logs:/var/log/kolla/"
      - "libvirtd:/var/lib/libvirt"
      - "nova_compute:/var/lib/nova/"
      - "nova_libvirt_qemu:/etc/libvirt/qemu"
  when: inventory_hostname in groups['compute']
```

容器启动脚本会根据nova-compute.json来将配置文件拷贝到/etc并设置合适的权限

```
{
    "command": "nova-compute",
    "config_files": [
        {
            "source": "{{ container_config_directory }}/nova.conf",
            "dest": "/etc/nova/nova.conf",
            "owner": "nova",
            "perm": "0600"
        }{% if nova_backend == "rbd" %},
        {
            "source": "{{ container_config_directory }}/ceph.*",
            "dest": "/etc/ceph/",
            "owner": "nova",
            "perm": "0700"
        }{% endif %}
    ]
}
```

关于kolla配置文件的管理还可以参考这里

nova-fake测试控制平台性能

这里

compute节点升级问题

由于所有服务都运行在容器中，那么是不是我升级compute节点时，该节点的虚机都会进入关机状态呢，kolla使用super-privilege的容器来解决了这个问题具体可以参考kolla PTL的文章https://sdake.io/2015/01/28/an-atomic-upgrade-process-for-openstack-compute-nodes/

平滑升级

kolla为升级也编写了upgrade.yaml这个playbook,我们还是拿nova-compute的升级为例

```
# kolla/ansible/roles/nova/tasks/upgrade.yml
---
# Create new set of configs on nodes
- include: config.yml

# TODO(inc0): since nova is creating new database in L->M, we need
# It should be removed later
- include: bootstrap.yml

- include: bootstrap_service.yml

- name: Checking if conductor container needs upgrading
  kolla_docker:
    action: "compare_image"
    common_options: "{{ docker_common_options }}"
    name: "nova_conductor"
    image: "{{ nova_conductor_image_full }}"
  when: inventory_hostname in groups['nova-conductor']
  register: conductor_differs

# Short downtime here, but from user perspective his call will just
- name: Stopping all nova_conductor containers
  kolla_docker:
    action: "stop_container"
    common_options: "{{ docker_common_options }}"
    name: "nova_conductor"
  when:
    - inventory_hostname in groups['nova-conductor']
    - conductor_differs['result']

- include: start_conductors.yml

- include: start_controllers.yml
  serial: "30%"

- include: start_compute.yml
  serial: "10%"

- include: reload.yml
```

```
  serial: "30%"
```

使用

查看log

```
cd /var/lib/docker/volumes/kolla_logs/
```

进入容器调试

```
docker exec -it service_name  bash
```

root权限问题

出于安全考虑很多kolla服务都是运行在非root下，进入容器后拿不到root权限，我们还以nova_compute为例,可以修改/etc/kolla/nova_compute/config.json改为以下

```
{
    "command": "nova-compute",
    "config_files": [
        {
            "source": "/var/lib/kolla/config_files/nova.conf",
            "dest": "/etc/nova/nova.conf",
            "owner": "nova",
            "perm": "0600"
        },
        {
            "source": "/var/lib/kolla/config_files/nova.sudo",
            "dest": "/etc/sudoers.d/nova.sudo",
            "owner": "root",
        }    ]
}
```

然后在/etc/kolla/nova-compute添加nova.sudo

```
nova        ALL=(ALL)       NOPASSWD: ALL
```

重启容器后即可sudo到root用户下调试

定制化build镜像

参考 https://github.com/openstack/kolla/blob/master/doc/image-building.rst

总结

优点

- 配置管理灵活方便
- 可以平滑升级
- 部署简单
- 环境隔离
- 多种安装源
- 支持的部署的服务多

缺点

- 对新手的友好程度
- debug不方便

TripleO

一、TripleO简介

TripleO 又叫 OpenStack on OpenStack，是一个用OpenStack来部署、升级和管理OpenStack的工具。 TripleO里面有两个主要部分：Undercloud和Overcloud，使用TripleO，你需要先创建一个小的OpenStack环境，称为:Undercloud，它包含了Heat、Ironic、Horizen、Keystone、Neutron等项目，TripleO使用Ironic做裸机管理，Heat做编排，Keystone做用户管理等，使用Undercloud部署出来的是正式的OpenStack环境：Overcloud。

二、TripleO部署需求

硬件需求

部署TripleO至少需要3个节点，角色分别如下：

- 一个Undercloud节点
- 一个Overcloud控制节点
- 一个Overcloud计算节点

最小配置：

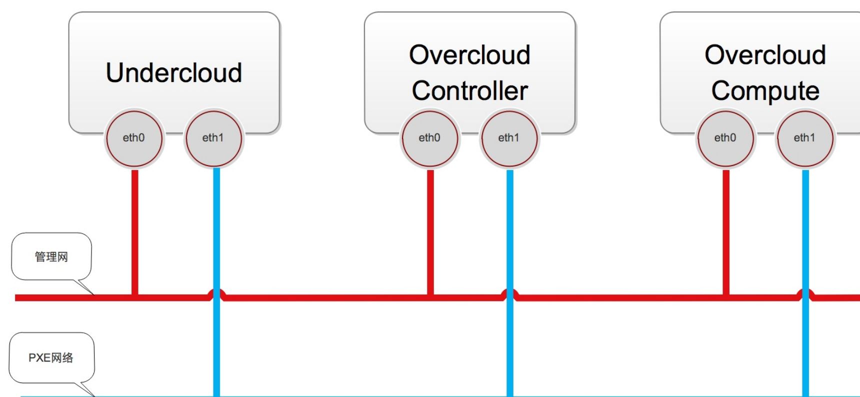
- 多核CPU
- 8G 内存
- 60G 硬盘

推荐使用物理机，开发测试也可以使用虚拟机，TripleO部署目前只支持RHEL 7.1 x86_64和CentOS 7 x86_64。

网络需求

> 注：此网络需求不包含Overcloud中OpenStack所需网络。
>
> - Overcloud服务器需要配置好IPMI。
> - 一个管理网卡，用于管理Undercloud和Overcloud。
> - 一块用于PXE的网卡，这个网卡在Overcloud中需要是同一名称的网卡，如em2，这个名称将会在安装配置的LOCAL_INTERFACE参数中使用，并且不能和管理网络是同一块网卡。
> - 在BIOS中，把用于PXE的网卡放到引导顺序的第一位，并且关闭除了PXE网卡之外所有网卡的网络启动选项。
> - 收集所有Overcloud节点的用于PXE引导网卡的MAC地址和IPMI信息。

网络参考下图：



TripleO提供了一个工具tripleo-validations用于检测TripleO部署可能出现的问题，该工具基于Ansible和Python编写，在validations目录下提供了很多检测的内容，如：

- NTP配置
- DHCP获取
- Pacemaker状态
- Undercloud硬件配置
- MySQL打开连接数限制

- ……

三、部署TripleO

由于TripleO也是一套小型的OpenStack环境，部署起来也较繁琐，于是出现了几个项目用来部署TripleO，如：Tripleo-Quickstart和官方推荐项目Instack-Undercloud。 其中,Tripleo-Quickstart主要使用Ansible部署、Instack-Undercloud则混合使用了脚本和Puppet来部署，但是使用Puppet的时候有些调用模块的时候会出问题，调用的资源改名时Instack-Undercloud里没有修改，所以在使用Instack-Undercloud部署TripleO的时候最好了解一些Puppet知识。 另外，由于使用RDO的源要下载很多包，网速慢的同学就要痛苦了，可能要部署很长时间。

四、TripleO部署OpenStack

1. 生成镜像

TripleO部署需要如下几个镜像：

```
ironic-python-agent.initramfs
ironic-python-agent.kernel
overcloud-full.initrd
overcloud-full.qcow2
overcloud-full.vmlinuz
```

> 注：如果只是测试的话也可以从CentOS网站上下载

如果使用的是CentOS系统，可以直接使用以下命令创建： `openstack overcloud image build` ，而如果是RHEL系统，则需要通过 `--config-file` 参数进行指定配置文件， `openstack overcloud image build --config-file /usr/share/openstack-tripleo-common/image-yaml/overcloud-images.yaml --config-file $OS_YAML`

> 注：image-yaml目录是在5.0版本后才有的，Mitaka版本装的是2.1.1版本，并没有这个目录。

2. 上传镜像

使用如下命令上传镜像`openstack overcloud image upload`,更新镜像使用`openstack overcloud image upload --update-existing`，如果上传的是.initramfs后缀的镜像，需要执行以下命令重新配置Ironic使用这个镜像.
`openstack overcloud node configure --all-manageable`

3. 注册节点

注册节点是注册到Ironic里面，我们在前面提过，TripleO使用Ironic管理裸机，`openstack overcloud node import instackenv.json`，使用的文件可以是JSON、YAML或CSV文件，导入时根据后缀名判断，该JSON文件模板如下：

```
{
    "nodes": [
        {
            "pm_type":"pxe_ipmitool",
            "mac":[
                "fa:16:3e:2a:0e:36"
            ],
            "cpu":"2",
            "memory":"4096",
            "disk":"40",
            "arch":"x86_64",
            "pm_user":"admin",
            "pm_password":"password",
            "pm_addr":"10.0.0.8"
        },
        ……省略两个节点
    ]
}
```

该命令只会导入信息，不会进行检查，导入后状态为：manageable，Ironic中服务器的状态有以下几种：

- enroll，该状态是Ironic不会对其进行管理，在Newton版本后，支持使用该状态替换available状态，即导入时添加`--initial-state=enroll`参数。
- manageable，验证完IPMI等可用后，服务器被设置为manageable状态，在这个状态时，用户可以进行自检、RAID配置等操作，但还不能开始部署。
- available，部署前的最后一个状态，此状态时Ironic可以随时开始部署。

如果要在导入时进行检查，则执行以下命令： `openstack overcloud node import --introspect --provide instackenv.json`

4. 节点自检

上节说过，在manageable状态时，用户可以执行自检程序，命令如下: Mitaka版本中则是使用 `openstack baremetal introspection bulk start` ，之后的版本使用： `openstack overcloud node introspect --all-manageable` ，如果只需要检查一个节点可以先把该节点置为manageable状态，然后执行检查:

```
ironic node-set-provision-state UUID manage
openstack baremetal introspection start UUID
```

然后使用以下命令查看这台机器的状态，看其中的**finished**是否为**True**： `openstack baremetal introspection status UUID`

5. 部署节点

使用 `openstack overcloud deploy` 命令部署，后面使用 `—templates` 指定的Heat模板， `-e` 指定Heat环境文件，给Heat传递参数，由于TripleO使用Heat部署Overcloud，理所当然，你起码得了解Heat，知道该怎么用，附录中有Heat模板的连接，感兴趣的可以看一下，这里我们就不多做介绍了。 TripleO支持控制节点HA部署,至少需要3个节点通过Pacemaker实现HA功能，通过以下方式添加到deploy命令的后面：

```
# cat << EOF > ~/environment.yaml
parameter_defaults
  ControllerCount: 3
EOF

# openstack overcloud deploy -e environment.yaml -e /usr/share/oper
```

6. 总结

优点：

- 熟悉OpenStack的人使用方便
- 很好的管理物理服务器，完整的生命周期管理

缺点

- 部署麻烦
- 上手较困难

参考

Heat模板：https://docs.openstack.org/developer/heat/template_guide/index.html

Packstack

简介

Packstack主要是由Redhat推出的用于概念验证（PoC）环境快速部署的工具。Packstack是一个命令行工具，它使用Python封装了Puppet模块，通过SSH在服务器上部署OpenStack。

Packstack支持三种运行模式：

- 快速运行
- 交互式运行
- 非交互式运行

Packstack支持两种部署架构：

- All-in-One，即所有的服务部署到一台服务器上
- Multi-Node，即控制节点和计算机分离

因为Redhat官方有详细的使用文档，因此本文将简要介绍Packstack的快速运行以及交互式运行方式来部署All-in-One的Openstack。

部署前准备

在开始部署前，我们需要准备一台虚拟机，它的规格如下：

名称	要求
处理器	推荐2核以上
内存	推荐4G以上
磁盘	推荐20G以上
网卡	至少一块1G网卡
操作系统	CentOS7.2

在完成虚拟机的配置和启动后，在终端下输入以下指令：

```
$ sudo yum install -y https://www.rdoproject.org/repos/rdo-release
$ sudo yum update -y
$ sudo yum install -y openstack-packstack
```

快速运行

快速运行模式，表示用户可以对参数不做任何配置即可开始部署，用户只需要决定是单节点还是多节点的部署方式。

单节点

在packstack命令后，使用--allinonec参数在本机上部署所有服务。

```
$ packstack --allinone
```

多节点

使用--install-hosts参数来运行packstack，该参数值是由一个逗号隔开的IP地址列表。

```
$ packstack --install-hosts=CONTROLLER_ADDRESS,NODE_ADDRESSES
```

Packstack在部署完成后在终端上会输出以下信息：

```
 **** Installation completed successfully ******
```

交互式运行

1.如果希望以交互式的方式来配置集群的部署，可以在终端下输入：

```
# packstack
```

2.packstack会提示你输入一个用于保存公共密钥的路径，输入 `Enter` ，则 会使用默认的 `~/.ssh/id_rsa.pub` ：

```
Enter the path to your ssh Public key to install on servers:
```

3.packstack提示输入一个默认密码，该密码将作为admin user密码， 不输入则随机生成：

```
Enter a default password to be used. Leave blank for a randomly ger
```

4.输入每个wsgi服务的进程数，默认等于cpu的核数：

```
Enter the amount of service workers/threads to use for each service
```

5.确认是否要安装MariaDB数据库,默认为y：

```
Should Packstack install MariaDB [y|n]  [y] :
```

6.确认是否安装Openstack组件，可以根据需要定制服务：

```
Should Packstack install OpenStack Image Service (Glance) [y|n]  [y
Should Packstack install OpenStack Block Storage (Cinder) [y|n]  [y
Should Packstack install OpenStack Shared File System (Manila) [y|n
Should Packstack install OpenStack Compute (Nova) [y|n]  [y] :
Should Packstack install OpenStack Networking (Neutron) [y|n]  [y]
Should Packstack install OpenStack Dashboard (Horizon) [y|n]  [y] :
Should Packstack install OpenStack Object Storage (Swift) [y|n]  [y
Should Packstack install OpenStack Metering (Ceilometer) [y|n]  [y]
Should Packstack install OpenStack Telemetry Alarming (Aodh) [y|n]
Should Packstack install OpenStack Resource Metering (Gnocchi) [y|n
Should Packstack install OpenStack Clustering (Sahara). If yes it'l
Should Packstack install OpenStack Orchestration (Heat) [y|n]  [n]
Should Packstack install OpenStack Database (Trove) [y|n]  [n] :
Should Packstack install OpenStack Bare Metal (Ironic) [y|n]  [n] :
Should Packstack install OpenStack client tools [y|n]  [y] :
```

7.Packstack为所有服务配置NTP服务来校准系统时间，NTP的设置只对多节点有意义：

```
Enter a comma separated list of NTP server(s). Leave plain if Packs
```

8.是否安装Nagios监控服务：

```
Should Packstack install Nagios to monitor OpenStack hosts [y|n]  |
```

9.哪些服务器在本次安装中被排除在外：

```
Enter a comma separated list of server(s) to be excluded. Leave pla
```

10.是否启用调试模式：

```
Do you want to run OpenStack services in debug mode [y|n]  [n] :
```

11.指定控制器的地址:

```
Enter the controller host  [10.211.55.8] :
```

12.指定计算节点的地址：

```
Enter list of compute hosts  [10.211.55.8] :
```

13.指定网络节点的地址：

```
Enter list of network hosts  [10.211.55.8] :
```

14.是否使用VMWare vCenter作为hypervisor和datastore的后端：

```
Do you want to use VMware vCenter as hypervisor and datastore [y|n]
```

15.指定是否使用不支持的参数，推荐使用默认设置：

```
Enable this on your own risk. Do you want to use unsupported parame
```

16.网卡名称是否被自动识别为子网+CIDR的格式：

```
Should interface names be automatically recognized based on subnet
```

17.是否为每个服务器订阅Extra Packstacks for Enterprise Linux(EPEL)，建议使用默认设置：

```
To subscribe each server to EPEL enter "y" [y|n]  [n] :
```

18.是否启用自定义的软件包仓库:

```
Enter a comma separated list of URLs to any additional yum reposito
```

19.是否启用rdo test：

```
To enable rdo testing enter "y" [y|n]  [n] :
```

20.是否启用Red Hat订阅，跳过即可：

```
To subscribe each server to Red Hat enter a username :
To subscribe each server with RHN Satellite enter RHN Satellite ser
```

21.ssl证书相关的操作：

```
Enter the filename of the SSL CAcertificate, if the CONFIG_SSL_CACE
Enter the filename of the SSL CAcertificate Key file, if the CONFIG
Enter the path to use to store generated SSL certificates in  [~/pa
Should packstack use selfsigned CAcert. [y|n]  [y] :
Enter the ssl certificates subject country.  [--] :
Enter the ssl certificates subject state.  [State] :
Enter the ssl certificate subject location.  [City] :
Enter the ssl certificate subject organization.  [openstack] :
Enter the ssl certificate subject organizational unit.  [packstack]
Enter the ssl certificaate subject common name.  [centos-7.1.sharec
Enter the ssl certificate subject admin email.  [admin@centos-7.1.s
```

22.配置AMQP服务，默认会使用RabbitMQ作为backend，不启用身份验证和SSL：

```
Set the AMQP service backend [rabbitmq]  [rabbitmq] :
Enter the host for the AMQP service  [10.211.55.8] :
Enable SSL for the AMQP service? [y|n]  [n] :
Enable Authentication for the AMQP service? [y|n]  [n] :
```

23.配置MariaDB服务

```
Enter the IP address of the MariaDB server  [10.211.55.8] :
Enter the password for the MariaDB admin user :
Confirm password :
```

24.配置Identity服务，包括设置数据库连接的密码，创建默认的admin,demo用户等基本操作：

```
Enter the password for the Keystone DB access :
Confirm password :
Enter y if cron job for removing soft deleted DB rows should be cre
Confirm password [y|n]  [y] :
Region name  [RegionOne] :
Enter the email address for the Keystone admin user  [root@localhos
Enter the username for the Keystone admin user  [admin] :
Enter the password for the Keystone admin user :
Confirm password :
Enter the password for the Keystone demo user :
Confirm password :
Enter the Keystone identity backend type. [sql|ldap]  [sql] :
```

25.配置Image服务，包括设置数据库连接密码，glance用户密码，后端存储：

```
Enter the password for the Glance DB access :
Confirm password :
Enter the password for the Glance Keystone access :
Confirm password :
Glance storage backend [file|swift]  [file] :
```

26.配置块存储服务，包括设置数据库连接密码，cinder用户和密码:

```
Enter the password for the Cinder DB access :
Confirm password :
Enter y if cron job for removing soft deleted DB rows should be cre
Confirm password [y|n]  [y] :
Enter the password for the Cinder Keystone access :
Confirm password :
Enter the Cinder backend to be configured [lvm|gluster|nfs|vmdk|net
Should Cinder's volumes group be created (for proof-of-concept inst
Enter Cinder's volumes group usable size  [20G] :
Enter y if cron job for removing soft deleted DB rows should be cre
Confirm password [y|n]  [y] :
```

27.配置计算服务，包括flavor,资源虚拟比，迁移，虚拟化软件等参数的设置：

```
Should Packstack manage default Nova flavors [y|n]  [y] :
Enter the CPU overcommitment ratio. Set to 1.0 to disable CPU overc
Enter the RAM overcommitment ratio. Set to 1.0 to disable RAM overc
Enter protocol which will be used for instance migration [tcp|ssh]
Enter the compute manager for nova migration  [nova.compute.manager
Enter the path to a PEM encoded certificate to be used on the https
Enter the SSL keyfile corresponding to the certificate if one was e
Enter the PCI passthrough array of hash in JSON style for controlle
Enter the PCI passthrough whitelist as array of hash in JSON style
The nova hypervisor that should be used. Either qemu or kvm. [qemu
Confirm password [qemu|kvm]  [%{::default_hypervisor}] :
```

28.配置网络服务，包括从组件，接口，网络驱动等细节的设置：

```
Enter the password for Neutron Keystone access :
Confirm password :
Enter the password for Neutron DB access :
Confirm password :
Enter the ovs bridge the Neutron L3 agent will use for external tra
Enter Neutron metadata agent password :
Confirm password :
Should Packstack install Neutron LBaaS [y|n]  [n] :
Should Packstack install Neutron L3 Metering agent [y|n]  [y] :
Would you like to configure neutron FWaaS? [y|n]  [n] :
Would you like to configure neutron VPNaaS? [y|n]  [n] :
Enter a comma separated list of network type driver entrypoints [lo
Enter a comma separated ordered list of network_types to allocate a
Enter a comma separated ordered list of networking mechanism drive
Enter a comma separated  list of physical_network names with which
Enter a comma separated list of physical_network names usable for V
Enter a comma separated list of <tun_min>:<tun_max> tuples enumerat
Enter a multicast group for VXLAN:
Enter a comma separated list of <vni_min>:<vni_max> tuples enumerat
Enter the name of the L2 agent to be used with Neutron [linuxbridge
Enter a comma separated list of supported PCI vendor devices, defin
Set to y if the sriov agent is required [y|n]  [n] :
Enter a comma separated list of interface mappings for the Neutron
Enter a comma separated list of bridge mappings for the Neutron ope
Enter a comma separated list of OVS bridge:interface pairs for the
Enter a comma separated list of bridges for the Neutron OVS plugin
Enter interface with IP to override the default tunnel local_ip:
Enter comma separated list of subnets used for tunneling to make th
Enter VXLAN UDP port number  [4789] :
```

29.设置Dashboard服务，是否开启Https服务：

```
Would you like to set up Horizon communication over https [y|n]  [n
```

30.配置对象存储服务,包括设备逻辑，zone，replicas,文件系统和块设备大小的配置：

```
Enter the Swift Storage devices e.g. /path/to/dev:
Enter the number of swift storage zones, MUST be no bigger than the
Enter the number of swift storage replicas, MUST be no bigger than
Enter FileSystem type for storage nodes [xfs|ext4]  [ext4] :
Enter the size of the storage device (eg. 2G, 2000M, 2000000K)  [20
```

31.是否启用Tempest服务:

```
Would you like to provision for demo usage and testing [y|n]  [y] :
Would you like to configure Tempest (OpenStack test suite). Note th
```

32.设置Floating IP网段

```
Enter the network address for the floating IP subnet  [172.24.4.224
```

33.设置测试镜像的名称，源地址，格式等配置：

```
Enter the name to be assigned to the demo image  [cirros] :
Enter the location of an image to be loaded into Glance  [http://do
Enter the format of the demo image  [qcow2] :
Enter the name of a user to use when connecting to the demo image v
Enter the name to be assigned to the uec image used for tempest  [c
Enter the location of a uec kernel to be loaded into Glance  [http:
Enter the location of a uec ramdisk to be loaded into Glance  [http
Enter the location of a uec disk image to be loaded into Glance  [h
Would you like to configure the external ovs bridge [y|n]  [y] :
```

34.设置Ceilometer,Aodh,Gnocchi服务：

```
Enter the password for Gnocchi DB access :
Confirm password :
Enter the password for the Gnocchi Keystone access :
Confirm password :
Enter the password for the Ceilometer Keystone access :
Confirm password :
Enter the Ceilometer service name. [ceilometer|httpd]  [httpd] :
Enter the host for the MongoDB server  [10.211.55.8] :
Enter the host for the Redis server  [10.211.55.8] :
Enter the port of the redis server(s)  [6379] :
Enter the password for the Aodh Keystone access :
Confirm password :
```

35.设置nagios用户的密码:

```
Enter the password for the nagiosadmin user :
```

36.最后一步，确认生成的配置是否符合期望，输入 `yes` ，并按 `回车` 键开始执行操作：

```
Packstack will be installed using the following configuration:
==============================================================
ssh-public-key:                /root/.ssh/id_rsa.pub
default-password:
service-workers:               %{::processorcount}
mariadb-install:               y
......
aodh-ks-passwd:                ********
nagios-passwd:                 ********
Proceed with the configuration listed above? (yes|no):
```

非交互式方式运行

使用下述命令生成一个answer file:

```
# packstack --gen-answer-file=my_file
```

使用vim打开文件,每个配置项都含有详细的说明：

```
[general]

# Path to a public key to install on servers. If a usable key has r
# been installed on the remote servers, the user is prompted for a
# password and this key is installed so the password will not be
# required again.
CONFIG_SSH_KEY=/root/.ssh/id_rsa.pub

# Default password to be used everywhere (overridden by passwords s
# for individual services or users).
CONFIG_DEFAULT_PASSWORD=

# The amount of service workers/threads to use for each service.
# Useful to tweak when you have memory constraints. Defaults to the
# amount of cores on the system.
CONFIG_SERVICE_WORKERS=%{::processorcount}

# Specify 'y' to install MariaDB. ['y', 'n']
CONFIG_MARIADB_INSTALL=y

# Specify 'y' to install OpenStack Image Service (glance). ['y', 'r
CONFIG_GLANCE_INSTALL=y

......
```

例如，我们不希望配置MariaDB，只需要将 `CONFIG_MARIADB_INSTALL` 设置为 `n` ：

```
CONFIG_MARIADB_INSTALL=n
```

保存并退出my_file，在终端下运行以下命令指定相应的配置文件：

```
# packstack --answer-file=my_file
```

深入理解Packstack

Packstack的使用非常简单，关于如何使用的介绍就到此结束。接下来才是重点，我们要深入到Packstack的核心逻辑: Plugin，并且举例说明如何编写Plugin来完成对Packstack的功能扩展。

什么是Plugin

在前面两章对于PuppetOpenstack modules的介绍中，所有服务的部署工作实际是由每个modules完成的。 在使用Packstack的时候，我们发现Packstack支持大量的服务部署，例如:nova,glance,maridb,amqp等等。在其背后每个服务的配置项管理都是由plugin实现的，其路径是在: packstack/plugins，它看起来是这样的：

- **init**.py
- amqp_002.py
- aodh_810.py
- ...
- trove_850.py

每个plugin的名称都是由服务名称+下划线+三位数字编码组成，那么这些数字有什么作用？

我们来看一下packstack代码入口 `packstack/installer/run_setup.py` 是怎么加载plugins的:

在主函数入口，可以看到第一步调用了loadPlugins函数来加载插件：

```
def main():
    options = ""

    try:
        # Load Plugins
        loadPlugins()
        initPluginsConfig()
```

接着，我们跳转到了loadPlugins函数的定义，可以看到其中使用了sorted函数对由plugin文件组成的列表进行排序：

```
def loadPlugins():
    """
    Load All plugins from ./plugins
    """
    sys.path.append(basedefs.DIR_PLUGINS)
    sys.path.append(basedefs.DIR_MODULES)

    fileList = [f for f in os.listdir(basedefs.DIR_PLUGINS) if f[0]
    fileList = sorted(fileList, cmp=plugin_compare)  #使用plugin_co
    for item in fileList:
        # Looking for files that end with ###.py, example: a_plugin
        match = re.search("^(.+\_\d\d\d)\.py$", item)
        if match:
            try:
                moduleToLoad = match.group(1)
                logging.debug("importing module %s, from file %s",
                moduleobj = __import__(moduleToLoad)
                moduleobj.__file__ = os.path.join(basedefs.DIR_PLUG
                globals()[moduleToLoad] = moduleobj
                checkPlugin(moduleobj)
                controller.addPlugin(moduleobj)
            except:
                logging.error("Failed to load plugin from file %s",
                logging.error(traceback.format_exc())
                raise Exception("Failed to load plugin from file %s
```

查看函数plugin_compare的定义，我们终于找到了关键，plugin_compare使用每个plugin文件尾缀的三位数字用于排序比较：

```
def plugin_compare(x, y):
    """
    Used to sort the plugin file list
    according to the number at the end of the plugin module
    """
    x_match = re.search(".+\_(\d\d\d)", x)
    x_cmp = x_match.group(1)
    y_match = re.search(".+\_(\d\d\d)", y)
    y_cmp = y_match.group(1)
    return int(x_cmp) - int(y_cmp)
```

在了解了plugin的加载顺序后，我们再看看Plugin的代码结构。实际上，每个plugin的代码结构是一致的，由两个函数组成：

- `initConfig(controller)` 用于初始化Plugin的配置，主要是参数和参数组。
- `initSequences(controller)` 用于定义该plugin执行的任务。

在这些plugin中，必然会有一些与众不同的plugin，比如说第一个被加载的plugin，，倒数第二个被加载的plugin，以及最后一个被加载的plugin:

- `prescript_000.py` 是第一个被加载的plugin，顾名思义它提供了一些全局的初始化设置，比如ssh public key，default_password，workers的进程数量，是否开启各个OpenStack服务的设置等等，同时它会在被管理的主机上执行一些预备任务：生成authorized_keys文件，安装并开启epel源和rdo源，安装puppet软件包依赖和module依赖等等。
- `puppet_950.py` 是一个重要的plugin，顾名思义它提供了与puppet相关的任务，例如：生成最终的manifest文件，拷贝puppet modules到指定主机，生成hieradata文件，以standalone方式运行puppet：执行 `puppet apply` ，获取puppet运行中的输出等等。
- `postscript_951.py` 是最后一个呗加载的plugin，它只做了一件事情，就是运行Tempest跑测试任务。

动手写一个Plugin

在了解了plugin的运行机制后，我们来动手写一个plugin，我们称之为NOOP：这是一个空Plugin，默认只输出一行信息: NOOP Plugin.

创建一个Plugin文件

在packstack/plugins目录下，我们创建一个plugin文件: noop_840.py。

设置Import和Plugin定义

```
# -*- coding: utf-8 -*-

"""
Installs and configures NOOP
"""

from packstack.installer import basedefs
from packstack.installer import validators
from packstack.installer import processors
from packstack.installer import utils

from packstack.modules.common import filtered_hosts
from packstack.modules.documentation import update_params_usage
from packstack.modules.ospluginutils import generate_ssl_cert

# ------------- NOOP Packstack Plugin Initialization -------------

PLUGIN_NAME = "NOOP"
PLUGIN_NAME_COLORED = utils.color_text(PLUGIN_NAME, 'blue')
```

每个Plugin的import可能会有所不同，但大多数都会用到packstack.installer和packstack.modules。

此外，这里有两个和plugin相关的变量：

变量	说明
PLUGIN_NAME	plugin名称，全部大写字母
PLUGIN_NAME_COLORED	plugin显示的颜色，默认使用blue即可。

定义Plugin的配置信息

我们定义一个initConfig函数，其中包含了两个变量:

- params
- group

```
def initConfig(controller):
    params = [
                {"CMD_OPTION": "enable-noop",
                 "USAGE": "To set up noop service set this to 'y'",
                 "PROMPT": "Would you like to set up noop service",
                 "OPTION_LIST": ["y", "n"],
                 "VALIDATORS": [validators.validate_options],
                 "DEFAULT_VALUE": "n",
                 "MASK_INPUT": False,
                 "LOOSE_VALIDATION": True,
                 "CONF_NAME": "CONFIG_ENABLE_NOOP",
                 "USE_DEFAULT": False,
                 "NEED_CONFIRM": False,
                 "CONDITION": False},
             ]
    group = {"GROUP_NAME": "NOOP",
             "DESCRIPTION": "NOOP Config parameters",
             "PRE_CONDITION": "CONFIG_NOOP_INSTALL",
             "PRE_CONDITION_MATCH": "y",
             "POST_CONDITION": False,
             "POST_CONDITION_MATCH": True}
    controller.addGroup(group, params)
```

params是NOOP plugin定义的配置项，每个配置项的数据类型是字典。这些配置项可以作为顺序执行的一部分，或者作为Puppet模板的变量。

选项	说明
CMD_OPTION	被命令行使用的选项名称
USAGE	选项的使用说明，同时作为answer file的注释
PROMPT	交互模式下给用户的提示
OPTION_LIST	可选值列表，可以设置为[]或移除该选项，表示对选项值无限制
VALIDATORS	验证器函数列表，用于检查输入是否符合要求
DEFAULT_VALUE	选项的默认值
PROCESSORS	处理器函数列表，处理器函数对用户的输入做了处理，比如processors.process_host将主机名转变为IP地址等等
MASK_INPUT	是否隐藏用户的输入，如password
LOOSE_VALIDATION	若为true，则即使验证器返回为false，仍然使用用户输入的选项值
CONF_NAME	在answer file中的配置项名称，你同时可以在controller.CONF dict中找到
USE_DEFAULT	若为true,在交互模式下，将不会要求用户输入此变量的值，而直接DEFAULT_VALUE
NEED_CONFIRM	若为true，则要求用户确认其输入（比如password)
CONDITION	enable/disable该选项的条件,总是设置为False即可
DEPRECATES	弃用的CONF_NAME选项列表，通常在新版本时使用

group表示组的概念，在Packstack中，会把相关的配置项分组，这样就可以通过组的方式来管理和使用。group的数据类型是字典:

选项	说明
GROUP_NAME	组名，全局唯一
DESCRIPTION	组的描述，在命令行的帮助命令下会显示此信息
PRE_CONDITION	前提条件，可以是一个配置项的值或函数的返回值匹配预期。若为False，那么该配置组处于启用状态
PRE_CONDITION_MATCH	前提条件的预期匹配值
POST_CONDITION	配置组所有参数是正确的后置条件，若设置为False，则表示不做检查。通常设置为False
POST_CONDITION_MATCH	后置条件的预期匹配值，通常设置为True

这里最重要的是PRE_CONDITION和PRE_CONDITION_MATCH，可能有些晦涩，我们以部署Cinder服务为例，只有当PRE_CONDITION中的变量CONFIG_CINDER_INSTALL为"y"时，才会显示"Cinder"组的配置选项:

```
{"GROUP_NAME": "CINDER",
 "DESCRIPTION": "Cinder Config parameters",
 "PRE_CONDITION": "CONFIG_CINDER_INSTALL",
 "PRE_CONDITION_MATCH": "y",
 "POST_CONDITION": False,
 "POST_CONDITION_MATCH": True
}
```

最后一步，把这些已定义的选项添加controller的组中：

```
controller.addGroup(group, params)
```

定义函数执行顺序

前面我们说到每个plugin除了定义一组相关的选项之外，还会执行一些任务，比如：从用户给定的变量值来渲染template，从而生成该服务的Puppet manifest文件。这些任务是由一个个函数组成，函数之间有执行的先后顺序，这个顺序就是由initSeqeuence函数来决定。

我们假设NOOP服务的安装需要数据库服务MariaDB，以及在Keystone中创建endpoint等操作:

```
def initSequences(controller):
    if controller.CONF['CONFIG_NOOP_INSTALL'] != 'y':
    return
    steps = [{'title': 'Adding MariaDB manifest entries',
              'functions': [create_mariadb_manifest]},
             {'title': 'Adding NOOP manifest entries',
              'functions': [create_manifest]},
             {'title': 'Adding NOOP Keystone manifest entries',
              'functions': [create_keystone_manifest]}]
controller.addSequence('Installing NOOP service', [], [], steps)
```

setps是一个列表，其中每个元素的数据类型都是字典，它的格式如下：

选项	说明
title	函数的简单描述信息
functions	函数列表

最后，我们调用controller.addSequence()方法把plugin的steps添加到将要被执行的序列列表中。 通常情况下，第二个和第二个选项为空。

生成manifest文件

在讲生成manifest文件之前，我们需要花些时间来了解packstack的templates，在当前版本中packstack的templates的路径是packstack/puppet/templates，数量已经从几十个削减为3个:

- controller
- compute
- network

我们以controller为例，我们选取其中的代码片段:

```
stage { "init": before  => Stage["main"] }

Exec { timeout => hiera('DEFAULT_EXEC_TIMEOUT') }
Package { allow_virtual => true }

class {'::packstack::prereqs':
  stage => init,
}

include ::firewall

if hiera('CONFIG_NTP_SERVERS', '') != '' {
  include '::packstack::chrony'
}

include '::packstack::amqp'
include '::packstack::mariadb'

if hiera('CONFIG_MARIADB_INSTALL') == 'y' {
  include 'packstack::mariadb::services'
} else {
  include 'packstack::mariadb::services_remote'
}
```

我们可以发现，这实质上是一个manifest文件(.pp)，而非template文件(.erb)，所有class或define的include不再使用erb模板的方式来渲染，而是使用了简单的条件判断来做选择。

每个函数都是接受固定的两个参数:

- config
- messages

Openstack-Ansible

OSA简介

OpenStack-Ansible 是 OpenStack 社区的官方项目，可以部署在指定物理机器上，而不使用容器技术，从而可以更加容易的去的运维、升级、及扩展集群。同时OpenStack-Ansible（OSA）使用Ansible 自动化工具在Ubuntu Linux上部署OpenStack集群。为了隔离和易于维护，OSA也可以使用Linux容器（LXC）将Openstack核心服务安装到容器当中。除了OSA社区中官方项目Kolla，它的原理基于容器作为服务载体，使用Docker技术，如果有兴趣读者可以阅读Kolla章节。

- 注意了解本章需要你有一定的 Ansible 和 OpenStack的基本概念。本次我们通过一个 OpenStack-Ansible部署一套AIO环境

Ansible

Ansible是目前市面上非常流行一个自动化工具，主要是为了简化系统和应用程序的部署。Ansible通过SSH技术链接到每台服务器上。Ansible使用以YAML语言编写的手册进行编排。

Linux containers (LXC)

Linux Container容器是一种内核虚拟化技术。容器通过增强chroot环境的概念提供操作系统级虚拟化。容器为特定的一组进程隔离资源和文件系统，而没有虚拟机的开销和复杂性。们访问底层主机上的相同内核，设备和文件系统，并提供围绕一组规则构建的精简操作层。

OSA可配置的组件

可以部署Infrastructure 组件包括

- MariaDB with Galera
- RabbitMQ
- Memcached
- Repository
- Load balancer
- Utility container
- Log aggregation host
- Unbound DNS container

可以部署的Openstack组件包括 OpenStack services

- Bare Metal (ironic)
- Block Storage (cinder)
- Compute (nova)
- Container Infrastructure Management (magnum)
- Dashboard (horizon)
- Data Processing (sahara)
- Identity (keystone)
- Image (glance)
- Networking (neutron)
- Object Storage (swift)
- Orchestration (heat)
- Telemetry (aodh, ceilometer, gnocchi)

OSA 安装Openstack

环境准备,需要准备如下配置： *8 vCPU's* 50GB free disk space on the root partition *8GB RAM

最小化配置需求：

- CPU主板支持hardware-assisted的虚拟化
- 8 CPU Cores
- 80GB的主分区，或者60GB的第二块分区。（如果使用第二块分区需要使用

bootstrap_host_data_disk_device参数）。更多的细节可以去看如下链接：

官方建议配置：

- CPU主板支持hardware-assisted的虚拟化
- 8 CPU Cores
- 80GB的主分区，或者60GB的第二块分区。（如果使用第二块分区需要使用 bootstrap_host_data_disk_device参数）。更多的细节可以去看如下链接：
- 16GB RAM
- 参考文献：http://docs.openstack.org/developer/openstack-ansible/developer-docs/quickstart-aio.html

开始构建AIO环境 通过OpenStack-Ansible部署一套AIO环境有四步曲，但是第一步是可选项，此步骤需要你自己完成。

- 环境配置
- 安装bootstrap和Ansible
- 初始化主机的bootstrap
- 运行Playbook

注意：当部署一套新的集群时，我们建议将当前机器的内核及软件包升级到最新的版本。并且如下的命令都是通过Root用户来运行。可以在虚拟机中执行AIO构建以进行演示和评估，但是虚拟机性能会很差。对于生产环境中建议为特定角色使用多节点方式。

本次部署环境使用Ubuntu 14.04 LTS，Ansible 版本为2.2.0，首先在节点所有将 Package 更新

```
$ sudo apt-get dist-upgrade
```

git clone最新版本的openstack-ansible

```
$ sudo su -
$ git clone https://github.com/openstack/openstack-ansible
$ cd /opt/openstack-ansible
```

根据自己需求来部署对应的Openstack集群版本

```
# # 显示所有的TAG号
# git tag -l

# # 本次我们使用Newton版本搭建一套AIO
# git checkout stable/newton
# git describe --abbrev=0 --tags

# # Checkout the latest tag from either method of retrieving the ta
# git checkout 15.0.0.0rc1
```

设定参数在部署过程中会使用

```
export BOOTSTRAP_OPTS="bootstrap_host_data_disk_device=vdb"
export ANSIBLE_ROLE_FETCH_MODE=git-clone
```

环境执行工具的安装和初始化

```
$ scripts/bootstrap-ansible.sh
```

执行命令环境准备

```
$ scripts/bootstrap-aio.sh
```

最后执行部署命令

```
$ scripts/run-playbooks.sh
```

安装过程需要一段时间才能完成，但这里有一些一般估计：

- 带SSD存储的裸机系统：30-50分钟
- 具有SSD存储的虚拟机：约45-60分钟
- 系统与传统硬盘：90-120分钟

一旦playbook完全执行，可以尝试在/etc/openstack_deploy/user_variables.yml中的各种设置更改，并且只运行单个剧本。例如，要运行Keystone服务的剧本，请执行

```
# cd /opt/openstack-ansible/playbooks
# openstack-ansible os-keystone-install.yml
```

重新构建环境 有时，销毁所有容器并重建AIO是最佳的方案。虽然最佳方案是AIO被完全破坏和重建，但这并不是最佳的方案。因此，可以执行以下操作

```
# cd /opt/openstack-ansible/playbooks

# 删除所有的LXC容器
# openstack-ansible lxc-containers-destroy.yml

# # On the host stop all of the services that run locally and not
# #  within a container.
# for i in \
       $(ls /etc/init \
         | grep -e "nova\|swift\|neutron\|cinder" \
         | awk -F'.' '{print $1}'); do \
    service $i stop; \
  done

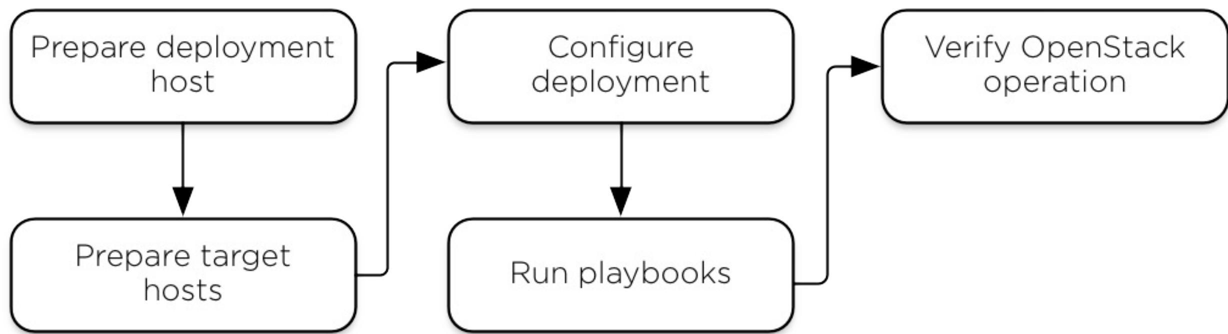
# # 卸除所有已经安装的Openstack服务
# for i in $(pip freeze | grep -e "nova\|neutron\|keystone\|swift\
    pip uninstall -y $i; done

# # 删除日志和配置目录
# rm -rf /openstack /etc/{neutron,nova,swift,cinder} \
         /var/log/{neutron,nova,swift,cinder}

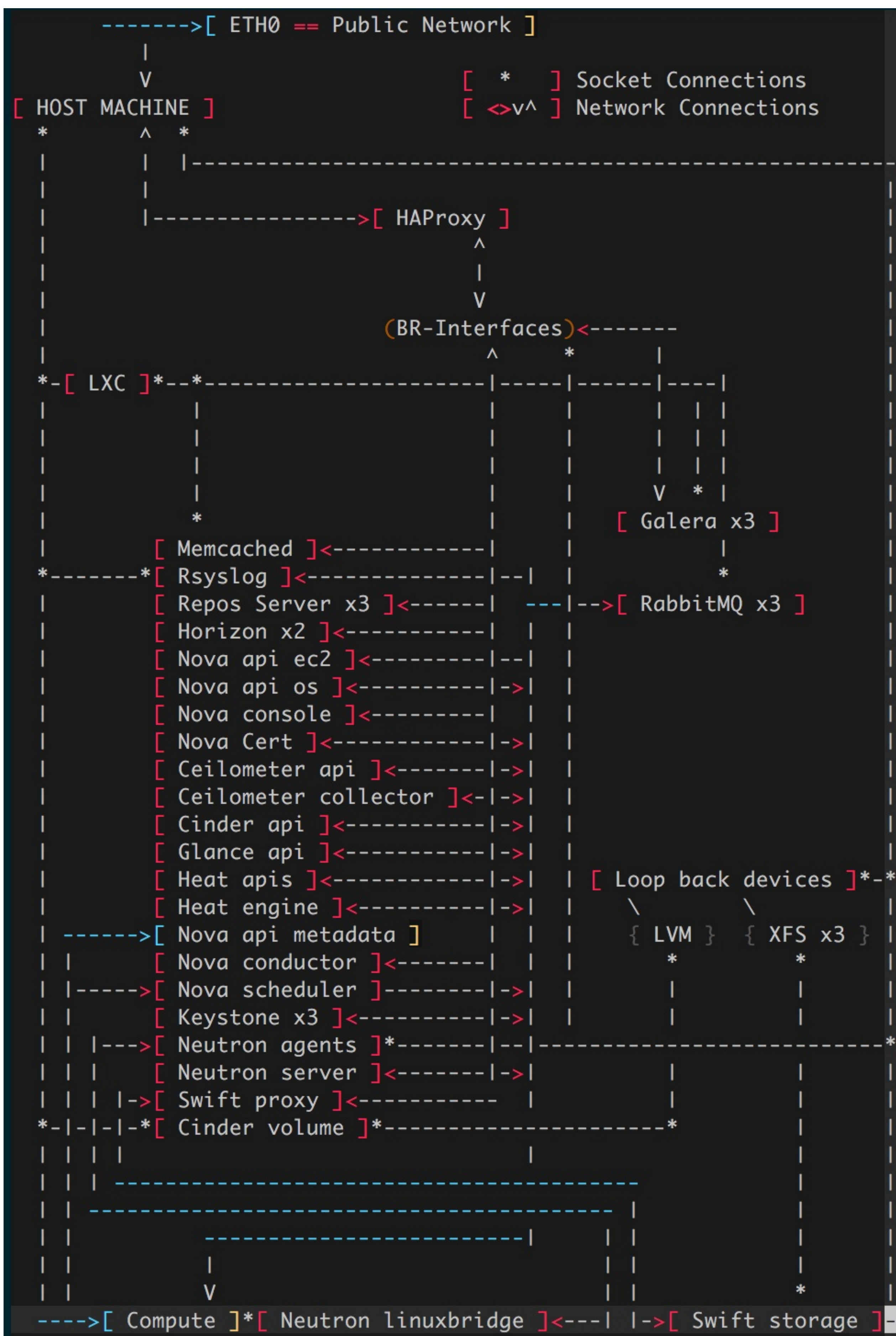
# # 删除pip配置文件
# rm -rf /root/.pip

# # 删除apt的proxy配置文件
# rm /etc/apt/apt.conf.d/00apt-cacher-proxy
```

OSA部署流程



OSA的AIO工作流程图


ETH0 == Public Network
HOST MACHINE
[*] Socket Connections
[<>v^] Network Connections
HAProxy
BR-Interfaces
LXC
Galera x3
Memcached
Rsyslog
Repos Server x3
RabbitMQ x3
Horizon x2
Nova api ec2
Nova api os
Nova console
Nova Cert
Ceilometer api
Ceilometer collector
Cinder api
Glance api
Heat apis
Loop back devices
Heat engine
Nova api metadata
LVM
XFS x3
Nova conductor
Nova scheduler
Keystone x3
Neutron agents
Neutron server
Swift proxy
Cinder volume
Compute
Neutron linuxbridge
Swift storage

OpenStack-ansible配置管理

OSA将安装服务文件存放在/etc/openstack_deploy/conf.d/目录当中，提供AIO和example展示当前主机组使用的文件。如果需要添加其它服务，分配主机到当前的主机文件当中，最后执行playbooks。

```
# 执行命令
# openstack-ansible setup-infrastructure.yml
```

setup-infrastructure.yml 文件包含了如下yaml

```
- include: unbound-install.yml
- include: repo-install.yml
- include: haproxy-install.yml
- include: memcached-install.yml
- include: galera-install.yml
- include: rabbitmq-install.yml
- include: etcd-install.yml
- include: ceph-install.yml
- include: utility-install.yml
- include: rsyslog-install.yml
```

Ansible 基础服务的playbooks安装如下基础服务：Memcached repository Galera RabbitMQ rsyslog memcached-install.yaml 详细代码，主要由两个role组成

```
- name: Install memcached
  hosts: memcached
  gather_facts: "{{ gather_facts | default(True) }}"
  max_fail_percentage: 20
  user: root
  pre_tasks:
    - include: common-tasks/os-lxc-container-setup.yml
    - include: common-tasks/os-log-dir-setup.yml
      vars:
        log_dirs:
          - src: "/openstack/log/{{ inventory_hostname }}-memcached
            dest: "/var/log/memcached"
    - include: common-tasks/package-cache-proxy.yml
  roles:
    - role: "memcached_server"
    - role: "rsyslog_client"
      rsyslog_client_log_rotate_file: memcached_log_rotate
      rsyslog_client_log_dir: "/var/log/memcached"
      rsyslog_client_config_name: "99-memcached-rsyslog-client.conf
      tags:
        - rsyslog
    - role: "system_crontab_coordination"
      tags:
        - crontab
  vars:
    is_metal: "{{ properties.is_metal|default(false) }}"
  tags:
    - memcached
```

role: "memcached_server" 默认参数

```
## Logging level
debug: False

## APT Cache Options
cache_timeout: 600

# Set the package install state for distribution packages
# Options are 'present' and 'latest'
memcached_package_state: "latest"

# Defines that the role will be deployed on a host machine
is_metal: true

# The default memcache memory setting is to use .25 of the availabl
# as long as that value is < 8192. However you can set the `memcach
# value to whatever you like as an override.
base_memcached_memory: "{{ ansible_memtotal_mb | default(4096) }}"
memcached_memory: "{{ base_memcached_memory | int // 4 if base_memc

memcached_port: 11211
memcached_listen: "127.0.0.1"
memcached_log: /var/log/memcached/memcached.log
memcached_connections: 1024
memcached_threads: 4
memcached_file_limits: "{{ memcached_connections | int + 1024 }}"

memcached_distro_packages: []
memcached_test_distro_packages: []
install_test_packages: False
```

更多参数可以参考:https://github.com/openstack/openstack-ansible-memcached_server

其它

安装前检查命令

```
openstack-ansible setup-hosts.yml --syntax-check
openstack-ansible setup-infrastructure.yml --syntax-check
openstack-ansible setup-openstack.yml --syntax-check
```

总结

优点

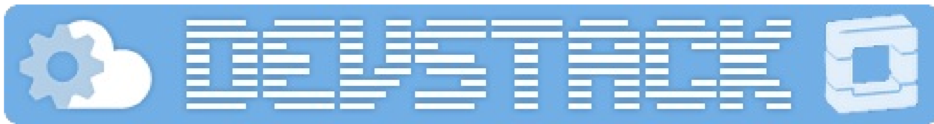
- 部署简单
- 支持的部署的服务多
- 可以自定义role中参数

缺点

- 友好程度

1.为什么需要Devstack ？

OpenStack是一个十分复杂的分布式系统，部署难度较大，调试也较困难。对于开发者，首先需要一个allinone的开发环境，可以随时修改代码并查看结果。各大厂商的部署工具一般都支持allinone的快速部署，比如红帽的RDO工具等。不过这些厂商的代码包通常是随着OpenStack的大版本发布而更新，不能实时与社区代码同步，而对于开发者而言，往往需要的是最新的代码，精确到最新的一次commit，因此使用厂商提供的部署工具难以满足开发需求。幸运的是社区已经提供了现成的快速部署工具，即DevStack(Develop OpenStack)，从英文名称上也能看出这是专为开发OpenStack量身打造的工具。



DevStack不依赖于任何自动化部署工具，纯Bash脚本实现，因此不需要花费大量时间耗在部署工具准备上，而只需要简单地编辑配置文件，然后运行脚本即可实现一键部署OpenStack环境。利用DevStack基本可以部署所有的OpenStack组件，但并不是所有的开发者都需要部署所有的服务，比如Nova开发者可能只需要部署核心组件就够了，其它服务比如Swift、Heat、Sahara等其实并不需要。DevStack充分考虑这种情况，一开始的设计就是可扩展的，除了核心组件，其它组件都是以插件的形式提供，开发者只需要根据自己的需求定制配置自己的插件即可。

DevStack除了给开发者快速部署最新的OpenStack开发环境，社区项目的功能测试也是通过DevStack完成，开发者提交的代码在合并到主分支之前，必须通过DevStack的所有功能集测试。另外，前面提到DevStack是基于代码仓库的master分支部署，如果你想尝试OpenStack的最新功能或者新项目，也可以通过DevStack工具快速部署最新代码的测试环境。

2. 三步玩转DevStack

刚刚提到DevStack的强大之处，是不是“蠢蠢欲动"想要小试牛刀？不过在开始之前，我得友情提醒下，DevStack运行后会安装大量OpenStack依赖的软件包和Python库，如果你怕弄乱你的系统，建议开一个虚拟机（你说用容器？you can，you up)，在虚拟机里跑DevStack就不用担心会弄坏你的系统了。目前DevStack支

持Ubuntu 14.04/16.04、Fedora 23/24、CentOS/RHEL 7以及Debian 和OpenSUSE操作系统，不过官方建议使用Ubuntu 16.04，因为该操作系统社区测试最全面，出现的问题最少。

OK，让我们开始一步步走起吧。

2.1 创建stack用户

为了系统的安全，DevStack最好不要在root用户下直接运行，因此需要创建一个专门的用户stack，该用户需要有免密码sudo权限，配置如下:

```
adduser stack
echo "stack ALL=(ALL) NOPASSWD: ALL" >> /etc/sudoers # 建议使用visud
su stack
```

如果已经下载了DevStack代码，DevStack也提供了一个专门的脚本创建stack用户，该脚本位于 `devstack/tools/create-stack-user.sh` ，直接运行该脚本即可。

最后请务必检查当前工作用户为stack，并且能够不输入密码执行sudo命令。

2.2 配置DevStack

在DevStack根目录下创建 `local.conf` 配置文件，包含admin密码、数据库密码、RabbitMQ密码以及Service密码：

```
[[local|localrc]]
ADMIN_PASSWORD=secret
DATABASE_PASSWORD=$ADMIN_PASSWORD
RABBIT_PASSWORD=$ADMIN_PASSWORD
SERVICE_PASSWORD=$ADMIN_PASSWORD
```

你也可以直接从 `sample` 目录下拷贝一个模板文件，然后在模板文件中修改。

部署一个最简单的OpenStack环境以上配置就够了，是不是特别简单？

2.3 Let DevStack Fly

```
./stack.sh
```

就这么简单？是的，一键部署，只需要一个命令！接下来你唯一需要做的，就是砌一杯咖啡静静地等待，取决于你的网络，通常需要等待半个小时。部署完后，会输出Dashboard地址以及默认创建的两个账号，一个是管理员账号admin，另一个是普通账号demo，如下：

```
This is your host IP address: 172.16.0.41
This is your host IPv6 address: ::1
Horizon is now available at http://172.16.0.41/dashboard
Keystone is serving at http://172.16.0.41:5000/
The default users are: admin and demo
The password: secret
```

注意以上部署的是一个精简版的OpenStack环境，默认只包含核心组件和Horizon，包括Keystone、Glance、Nova、Neutron、Cinder、Horizon等，其它服务则需要通过配置文件开启对应的插件完成，将在下面小节介绍。

2.4 关于下载速度优化

由于众所周知的原因，运行DevStack时下载OpenStack依赖包和Python库时非常慢，拉取OpenStack源码也非常耗时。为了避免这个问题，很多人都会从以下几个方面优化下载速度，加快部署速率：

1. 使用国内的镜像源

对于Ubuntu系统就是修改APT源，比如阿里云镜像源，只需要修改 `/etc/apt/source.list` 配置文件即可，替换为需要使用的镜像源。如：

```
deb http://mirrors.aliyun.com/ubuntu/ xenial main restricted univer
deb http://mirrors.aliyun.com/ubuntu/ xenial-security main restrict
deb http://mirrors.aliyun.com/ubuntu/ xenial-updates main restricte
deb http://mirrors.aliyun.com/ubuntu/ xenial-proposed main restrict
deb http://mirrors.aliyun.com/ubuntu/ xenial-backports main restric
deb-src http://mirrors.aliyun.com/ubuntu/ xenial main restricted ur
deb-src http://mirrors.aliyun.com/ubuntu/ xenial-security main rest
deb-src http://mirrors.aliyun.com/ubuntu/ xenial-updates main restr
deb-src http://mirrors.aliyun.com/ubuntu/ xenial-proposed main rest
deb-src http://mirrors.aliyun.com/ubuntu/ xenial-backports main res
```

2. 使用国内的pip源

只需要在当前家目录 `.pip` 目录创建 `pip.conf` 配置文件，以使用阿里云为例，配置文件内容如下：

```
cat ~/.pip/pip.conf
[global]
index-url = http://mirrors.aliyun.com/pypi/simple/
[install]
trusted-host=mirrors.aliyun.com
```

3.修改OpenStack源码地址

DevStack默认会从git.openstack.org下拉取代码，国内访问速度很慢，建议替换为github地址或者国内的trystack仓库，在 `[[local|localrc]]` 配置下增加以下配置项：

```
GIT_BASE=http://git.trystack.cn
```

如果你本地已经有最新的OpenStack源码了，也可以指定你本地的源码路径，比如使用本地的Nova源代码并且使用 `new_feature` 分支:

```
[[local|localrc]]
NOVA_REPO=/home/int32bit/nova
NOVA_BRANCH=new_feature
```

需要注意的是，国内源存在同步滞后，可能包不兼容或者下载某些包失败问题，出现这种情况时只需要重新替换原来的镜像源，然后重新运行 `./stack.sh` 即可。

3.使用DevStack环境开发

DevStack使用了Linux的终端复用工具screen，不同的服务运行在不同的window中，screen的使用方法可参考官方文档。通常情况下，我们都是针对OpenStack的某个组件进行开发，比如Nova，只需要找到Nova的源码路径，修改对应的源码，然后重启对应的服务即可。比如你修改了nova源码下的 `nova/compute/manager.py` 代码，则需要重启nova-compute服务，重启步骤如下：

- 使用 `screen -ls` 命令查找stack session。

```
# screen -ls
os3:~> screen -list
There is a screen on:
    28994.stack      (08/10/2016 09:01:33 PM)        (Detached)
1 Socket in /var/run/screen/S-sdague.
```

- 通过 `screen -r socket` attach到前台运行，其中 `socket` 为scrren的名称，以上为 `28994.stack` 。
- 使用 `ctrl-a n` 和 `ctrl-a p` 遍历windows，直到找到nova-compute服务。
- 使用 `ctrl-c` 杀掉nova-compute进程。
- 使用上下方向键遍历历史命令，找到跑nova-compute服务的命令，重新运行即可。

有些服务跑在Web服务器中，比如Keystone服务，此时重启Keystone服务只需要重启Apache服务即可。

如果你需要修改oslo代码或者python-xxxclient代码就相对麻烦点，因为这些代码不同于OpenStack源码，它默认不是从代码仓库中拉取，而是从已发布的pypi仓库中直接安装。你需要覆盖默认配置，手动配置代码仓库源:

```
[[local|localrc]]
LIBS_FROM_GIT=oslo.policy
OSLOPOLICY_REPO=/home/sdague/oslo.policy
OSLOPOLICY_BRANCH=better_exception
```

由于这些公共库需要被许多不同的项目依赖，因此社区的推荐做法是需要重新部署整个DevStack环境:

```
./unstack.sh && ./stack.sh
```

这个过程虽然不用重复从网络上下载包，相比第一次部署节省了不少时间，但仍然还是挺耗时间的，不利于单步调试。个人更倾向于改什么就重启什么服务，比如我修改了oslo.db代码，并且主要是解决Nova问题，那我只需要重启Nova服务即可，暂时不需要重新部署整个DevStack。而若修改了client代码，不需要重启任何服务，直接就可以测试功能。当然这种方式没有考虑其它服务的依赖，可能引入新问题，因此在确定开发完成后，最好还是完完全全走一遍unstack、stack流程。

4.使用DevStack部署其它OpenStack服务

前面我们使用DevStack部署了一个精简版的OpenStack环境，其中只包含了几个核心组件。其它OpenStack服务是通过插件形式安装，DevStack支持部署的所有插件列表可参考DevStack Plugin Registry，截至2017年2月份，DevStack共包含132个安装插件。其中包含：

- trove: 数据库服务。
- sahara: 大数据服务。
- ironic: 裸机服务。
- magnum: 容器编排服务。
- manila: 文件共享服务。
- cloudkitty: 计费服务。
- ...

需要开启部署某个服务，只需要使用 `enable_plugin` 配置指定对应插件即可，该配置项语法为:

```
enable_plugin plugin_name [code repo]
```

其中 `plugin_name` 为插件名称，可以在插件列表中找到， `code repo` 为代码仓库地址，不配置就使用默认的地址。

比如我们需要开启Sahara服务，只需要在 `local.conf` 增加以下配置项:

```
enable_plugin sahara https://github.com/openstack/sahara.git
enable_plugin sahara-dashboard https://github.com/openstack/sahara-
```

注意以上我们同时开启了两个Sahara相关的插件，前者是Sahara插件，而后者是dashboard的Sahara插件，若不配置该插件，在dashboard中将看不到Sahara面板。

除了OpenStack服务外，DevStack还支持其它和Openstack相关的插件，比如默认情况下都是使用本地文件系统存储作为OpenStack存储后端，如果需要测试Ceph后端，则需要开启devstack-plugin-ceph插件，该插件会自动部署一个单节点Ceph集群，然后就可以配置Glance、Nova、Cinder、Manila等服务使用Ceph后端了。

```
enable_plugin devstack-plugin-ceph git://git.openstack.org/openstac
ENABLE_CEPH_CINDER=True      # ceph backend for cinder
ENABLE_CEPH_GLANCE=True      # store images in ceph
ENABLE_CEPH_C_BAK=True       # cinder-backup volumes to ceph
ENABLE_CEPH_NOVA=True        # allow nova to use ceph resources
```

总结

本章首先介绍了DevStack的功能，然后详细介绍了如何使用DevStack快速部署一个OpenStack环境，最后介绍了使用DevStack部署其它OpenStack服务。DevStack项目是由社区维护的、专门为OpenStack开发人员准备的快速部署开发环境的脚本工具，该脚本工具具有非常灵活的扩展性，能够通过配置定制化部署

OpenStack服务。除此之外，社区项目的功能测试也是通过DevStack完成的。由此可见，DevStack是社区一个非常重要的项目，DevStack出现问题不仅影响开发者开发，还将可能导致社区的CI系统奔溃。DevStack使用Shell脚本实现，如果想学习Shell编程，DevStack源码将是一个很好的学习案例。

参考

1. DevStack官方文档: http://docs.openstack.org/developer/devstack/
2. Developing with Devstack: http://docs.openstack.org/developer/devstack/development.html
3. devstack ceph plugin: https://github.com/openstack/devstack-plugin-ceph

编写一个定制化的部署工具

需求和定制

在前面的篇幅中，我们介绍了各种部署工具和其使用的简单介绍。当你在使用它们来自动化地搭建企业的云平台时，会发现其实没有哪个工具可以完美地解决企业/用户的需求:

- 用户使用了一种新SDN技术，现有工具无法支持
- 云平台的架构比较特殊，现有工具无法支持
- 内部的认证/计费/报警系统也需要通过同一套部署工具进行整合，开源工具显然无法满足

那么在这种情况下，我们建议读者开始使用考虑编写一个定制化的部署工具，可以在现有工具的基础上进行二次开发，或者全新开发。

结语

近年来，我们看到越来越多的数据中心被各种技术云化，向外/内提供按需使用的服务。而OpenStack作为最流行的IaaS层开源项目，已经被运用到越来越多的数据中心中，而在这背后的运维自动化技术则发挥着举足轻重的作用。

软件部署是云数据中心运维workflow重要的开端，而后期为开发人员提供稳定的开发环境，提供与线上一致的测试环境，完成线上的计算/存储/网络/控制节点的扩容操作，完成服务的配置变更，完成软件版本的升级，提供安全加固等工作都是运维自动化的关注重点。

OpenStack在经历了多年的发展之后，已经演化出大量的部署自动化的项目/工具/产品，其中不少已经得到了生产环境的严格检验。

我们已经看到当前关于OpenStack自动化的趋势已从仅提供软件部署逐渐覆盖到物理机管理(如Fuel,TripleO)，与CI系统结合，提供用于运行集成测试的测试环境；提供Openstack服务的大版本升级能力，以及配置加固等安全能力。

我们庆幸出生在这个时代，因为我们正在改变数据中心，改变世界。

術語表

class

puppet中用于管理resource的集合，与面向对象编程中的类无关。

5.4. puppet-openstack-integration 5.9. puppet-stdlib
4.2. puppet-keystone模块 4.17. puppet-aodh模块 4.7. puppet-ceilometer
4.8. puppet-cinder 4.16. puppet-designate 4.5. puppet-glance
4.10. puppet-heat 4.6. puppet-horizon 4.1. OpenStack模块代码结构
4.14. puppet-manila 4.4. puppet-neutron 4.3. puppet-nova
5.1. puppet-oslo 4.15. puppet-rally 4.13. puppet-sahara 4.11. puppet-swift
4.9. puppet-tempest 4.12. puppet-trove 3.6. puppet-rabbitmq模块
3.1. puppet-apache模块 3.7. puppet-firewall模块
3.2. puppet-memcached模块 3.10. puppet-mongodb模块
3.8. puppet-mysql模块 3.4. puppet-rsync模块 3.3. puppet-sysctl模块
5.2. puppet-vswitch模块 6.5. 转发层规范 6.2. Hiera
1. 初识PuppetOpenstack 1.2. 术语表 7.4. Packstack 2.3. 理解Hiera
2.2. Puppet核心概念